Universitat de Lleida

TREBALL FINAL DE GRAU

Estudiant: Francisco Romero Batallé

Titulació: Grau en Enginyeria Informàtica

Títol de Treball Final de Grau: **Diseño e implementación de una librería para soportar de forma eficiente tipos de datos bioinformáticos en Spark**

Director/a: **Fernando Cores Prado**

Presentació

Mes: Setembre

Any: 2018

Universitat de Lleida
Escola Politècnica Superior

Universitat de Lleida

Trabajo Final de Grado

Diseño e implementación de una librería para soportar de forma eficiente tipos de datos bioinformáticos en Spark

Francisco Romero Batallé

Director:
Fernando Cores Prado

4 de septiembre de 2018

En agradecimiento a toda la gente que por poco o mucho ha hecho posible que este trabajo sea una realidad.

"Vive como si fueras a morir mañana. Aprende como si fueras a vivir siempre"
Mahatma Gandhi

Índice

Índice de figuras

Índice de tablas

Índice de listados de código

1. Introducción

Día a día se mueven cada vez volúmenes de datos más grandes por todo Internet. En muchos países se administran enormes bases de datos que contienen datos de censo de población, registros médicos, impuestos, etc., ademas de añadir transacciones financieras realizadas en línea o por dispositivos móviles, uso de redes sociales, elementos del Internet of Things (IoT), coordenadas GPS, y la suma de toda la actividad que se realiza durante el día con todos los dispositivos electrónicos ya sean smartphones u otro tipo, en la que es posible generar trafico de un total del orden de varios PetaBytes por día.

Esta tendencia ha abierto las puertas hacia un nuevo enfoque de entendimiento y toma de decisiones, la cual es utilizada para describir enormes cantidades de datos (estructurados, no estructurados, y semi estructurados) que tomaría demasiado tiempo y sería muy costoso cargarlos en una base de datos relacional para su análisis y procesado. De aquí nace el termino BigData [1], que aplica para toda aquella información que no puede ser procesada o analizada utilizando procesos o herramientas tradicionales sin referirse a ninguna cantidad en concreto, proponiendo una lista de características para entenderlo fácilmente, también llamadas como las 5-Vs [2] que detallaremos como:

- **Volumen**: referido al gran volumen de datos generados cada segundo, debido a la gran información que se genera en las redes sociales y las redes móviles, aunque el tiempo de vida de la información sea muy corta, cada vez se genera más, por lo que el uso que se estima que hay actualmente ya no se habla de TeraBytes, sino de ya del orden de PetaBytes y ZettaBytes.
- **Velocidad**: Referida sobre el tiempo en la que se van generando nuevos datos y a la velocidad que se mueven estos. Solo hay que pensar en la velocidad en la que se retransmite algún medio que se ha hecho viral por las redes sociales, o en la velocidad en la que se verifican transacciones de compra.
- **Variedad**: Se refiere a los diferentes tipos de datos que podemos usar. Estos pueden ser estructurados o no. En el pasado, casi todos los datos eran estructurados, pero actualmente hay un 80 % de datos no estructurados. Ademas de que el tipo a almacenar, tampoco es necesario que sea texto, también se incluyen: imágenes, mensajes de voz, vídeos, datos de sensores del IoT, conversaciones...
- **Veracidad**: Puede entenderse como el grado de confianza que les damos a los datos obtenidos, pudiendo establecer niveles de confianza para poder elegir que datos nos interesan mas según la fuente.
- **Valor**: En BigData podemos acumular mucha información, pero esta tiene que tener algún valor, ya que para poder sacarle provecho económico, hay que poder sacarle algún beneficio.

Para poder procesar esta información y obtener un valor añadido de ello no se puede hacer con las infraestructuras y los paradigmas y programas actuales. Se necesitan nuevas infraestructuras (Hadoop), nuevos frameworks (Spark) y nuevos paradigmas (MapReduce), de los que se pueden ver en las secciones posteriores más adelante. Tal como podemos observar en la *Figura* 1, hay diferentes categorizaciones sobre como enfocar BigData según el tipo de datos que vayamos a tratar. En nuestro caso elegiremos la rama de bioinformática, donde se puede observar un aumento del volumen de información usada y la necesidad de nuevos enfoques para su procesamiento en el de la Bionformática o Biologia Computacional.

Una de las nuevas tecnologías que han ayudado al reciente crecimiento de los datos ha sido las tecnologías de la secuenciación NGS (Next-Generation Sequencing) en el que se han podido reducir la velocidad y los costes de obtención para la secuenciación del ADN pudiendo obtener de forma paralela hasta 300 Gb de información sobre el ADN. Haciendo que el aumento de datos capturados cada vez sea mayor.

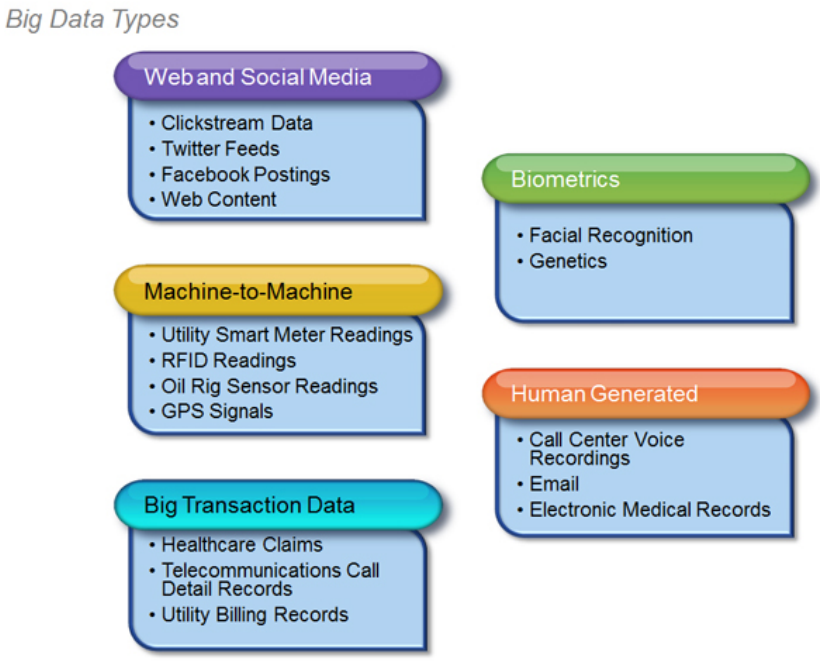

Figura 1: Tipos de datos en Big Data

Gracias a ello, actualmente se puede realizar una comparativa genómica con un mayor numero de datos y poderlo aplicar en medicina personalizada o de precisión. La cual permite realizar estudios personalizados sobre medicación a cada persona con los patrones del ADN obtenidos y almacenados en BigData, pudiendo obtener mejores resultados y elegir mejor medicación según la persona.
Como ya existe la disciplina de la biotecnología, en este proyecto nos enfocaremos en adaptar aplicaciones ya creadas, creando una nueva librería basada en Spark, para poder aplicar sobre un algoritmo generado por terceras personas y obtener grandes reducciones en el tiempo de computo.

1.1. Objetivos

El objetivo de este proyecto es utilizar los paradigmas, frameworks y las arquitecturas BigData para mejorar la eficiencia y las prestaciones de las aplicaciones bioinformáticas.

- Analizar los requisitos de las aplicaciones bioinformáticas.
- Estudiar y definir las técnicas de optimización más adecuadas para este tipo de aplicación.
- Elegir una aplicación como proyecto base y analizar sus necesidades para su optimización.
- Diseñar e implementar la librería independientemente a BDT-Coffee, lo cual implica:
  - Identificar y aislar las funciones usadas en BDT-Coffee para gestionar el almacenamiento de los datos (Como se guarda en disco, externamente, en una BD, etc.)
  - Extraer y definir las estructuras de datos en memoria para su procesamiento de datos.
  - Aislar el algoritmo de cálculo e implementarlo en nuestra librería.
  - Mejorar el algoritmo aislado, usando la paralelización que ofrece Apache Spark.
- Escalar y comparar el rendimiento utilizando nuestra librería.
- Generar un entorno de pruebas utilizando PPCAS como base para los Benchmarks:
  - Definir un conjunto de datos bioinformáticos y utilizar BDT-Coffee como base para Benchmark.

    - Introducir nuestra librería a BDT-Coffee y ejecutar nuevamente las pruebas con los mismos datos.
    - Analizar y comparar el rendimiento obtenido
- Documentar nuestra librería.
- Poder ofrecer mejoras a la comunidad científica

A partir de estos objetivos iniciales, durante el trascurso de la investigación, pueden ir apareciendo nuevas metas a medida que se vayan completando. Un objetivo que seria interesante completar, seria el de poder montar el proyecto en una plataforma externa de cloud computing, ya que se vería reflejado un hecho practico en el mundo real.

1.2. Motivación

Personalmente la principal motivación por la que he elegido este tema, es el poder aprovechar mi estancia en la universidad para poder aprender sobre tecnologías que no sean de ámbito doméstico, y sea necesario disponer de un hardware especifico o de unas tecnologías que no fuesen objeto de estudio para usuarios de nivel medio. Por lo que el utilizar el clúster que previamente habíamos usado en anteriores asignaturas era un buen punto de partida.

Una vez tenía claro la plataforma final donde poder trabajar, la intención era usar algún sistema distribuido donde se pueda escalar i obtener mejores resultados paralelizando el trabajo. Al principio, no tenia conocimiento de la existencia de Spark, pero si de Hadoop y el funcionamiento de MapReduce. Por lo tanto, al ver que Spark era como una sucesión de Hadoop, pero mucho mas potente y con bastantes características interesantes, se decidió como elección final Spark.

(a) Hadoop MapReduce

(b) Apache Spark

Figura 2: Tecnologías utilizadas

Finalmente solo faltaba decidir un tema a tratar de analizar referente a BigData, la elección directamente quería que se tratase sobre algún tema científico para poder aportar mi granito de arena y poder aportar algo para una comunidad científica en la que día a día va evolucionando, por lo que el tratar archivos sobre el genoma humano me pareció acertado y así al finalizar este proyecto, se espera poder dar un beneficio a la comunidad científica pudiendo entregar una librería optimizada y funcional para cualquier persona que esté interesada, ya que estará abierto de cara al publico en un repositorio en GitHub.

1.3. Presupuesto

Para la elección de este presupuesto existirá un precio para *investigación*, donde se le dedicará tiempo para tratar con las nuevas tecnologías que no se habían tocado antes. Una vez se haya estudiado la tecnología por completo, se pasará costear el presupuesto de *desarrollo*. Aquí es donde se desarrollará la librería en cuestión, e intentando obtener los mejores resultados posibles. De forma opcional se intentará desplegar la librería creada, sobre un sistema distribuido como puede ser el de docencia o en servicios de computación en la nube.

A continuación se muestra en la Tabla 1 un presupuesto aproximado con los puntos comentados anteriormente. Cabe destacar que son precios y horas aproximadas, haciendo un total de unas 350

horas de las 375 que hay establecidas para dedicar para el *TFG*, las 25 horas faltantes, serán para poder probar nuestra librería en un entorno de validación.

	Cantidad	**Precio**	**Total**
Coste derivados en investigación	100 h	30 €/ h	3.000 €
Coste necesarios en desarrollo	200 h	20 €/ h	4.000 €
Costes de implementación sistema	50 h	20 €/ h	1.000 €
Alquiler entorno validación (Cluster BigData, cloud)	25 h	Gratis	Gratis

Tabla. 1: Presupuesto

1.4. Planificación

Primeramente intentaremos establecer inicialmente como fecha límite el mes de *Julio*, por lo que se intentará hacer una planificación acorde para esta fecha. En caso de que no sea posible la entrega en esta fecha, o se vea conveniente que se aplace la entrega mas tarde, la fecha límite sería en Septiembre.

Para la realización de este proyecto tendremos que diferenciar entre tres etapas: la etapa de investigación, etapa de desarrollo, la etapa de validación y sintonización sobre un sistema distribuido y la etapa de documentación final, por lo que se ha definido el siguiente diagrama de GANTT en la figura 3

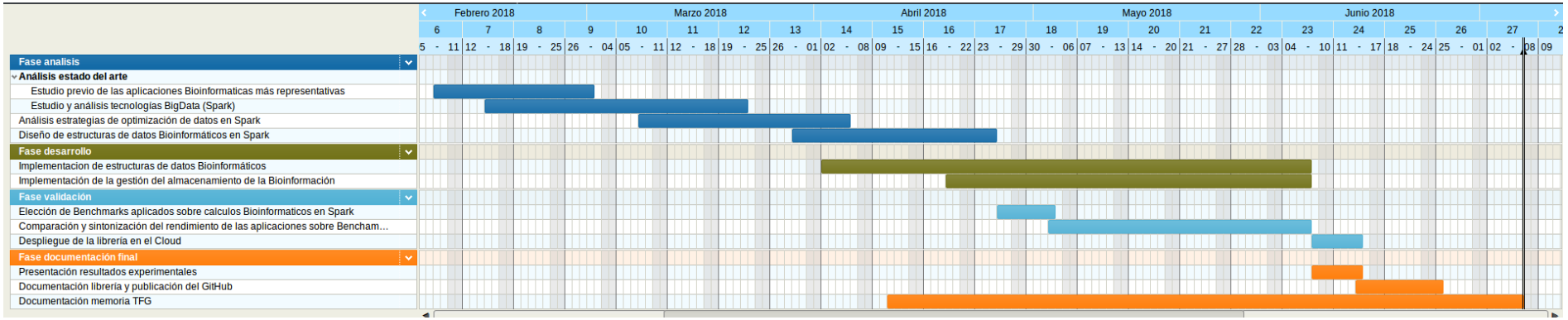

Figura 3: Diagrama GANTT

1.5. Estructura de la memoria

A continuación se resumirán los objetivos que deberá contener cada capítulo y cual será la estructura a seguir durante el proyecto.

Capítulo 2: Estado del arte
Sección donde se examinan las actuales tecnologías disponibles dentro de nuestro ámbito de búsqueda. Se hace un estudio sobre ellas para poder valorar si serán viables para usar en en la fase de *análisis y diseño*.

Capítulo 3: Análisis y Diseño
En esta sección del proyecto, se analizaran las diferentes tecnologías que podemos aprovechar para poder desarrollar correctamente nuestro proyecto en base a los requisitos que tengamos y analizar algunas de ellas.

Capítulo 4: Desarrollo
En este capítulo, nos basaremos en realizar los diagramas necesarios para el desarrollo y la implementación en base a los resultados obtenidos sobre las tecnologías analizadas en el capítulo anterior.

Capítulo 5: Resultados/Experimentación
Apartado del proyecto donde se indicarán resultados obtenidos, y que experimentos se han llevado a cabo para poder comparar resultados.

Capítulo 6: Conclusiones y trabajo futuro
En esta sección final del proyecto se comentarán las conclusiones que se han obtenido al final del transcurso del proyecto, y comentar como se podría continuar desarrollando la aplicación para obtener mejores resultados.

2. Estado del arte

En este capítulo del proyecto primero vamos a introducir las tecnologías BigData existentes, describiendo su función y ventajas ya que van a poder ser utilizadas en el proyecto. Seguidamente, también se comentarán algunas aplicaciones bioinformáticas, de las cuales, se ha llevado un estudio detallado sobre cada una de ellas, para que pueda ser utilizada como proyecto base para ser mejorada en nuestro proyecto. Por último y como parte del capítulo del estado del arte, se comentarán los distintos tipos de formatos de datos bioinformáticos, y su tratamiento para migrar a BigData y proceder a su gestión.

2.1. MapReduce

Aplicaciones científicas y de ingeniería a gran escala, y auditoria en la nube generan grandes cantidades de datos. El marco de MapReduce junto con la computación en la nube se están convirtiendo en la solución viable para el procesamiento de BigData distribuido, por lo que es necesario generar algún modelo de programación y una implementación para procesar y generar conjuntos de datos grandes con un algoritmo que sea paralelo y poder ser distribuido en un clúster.

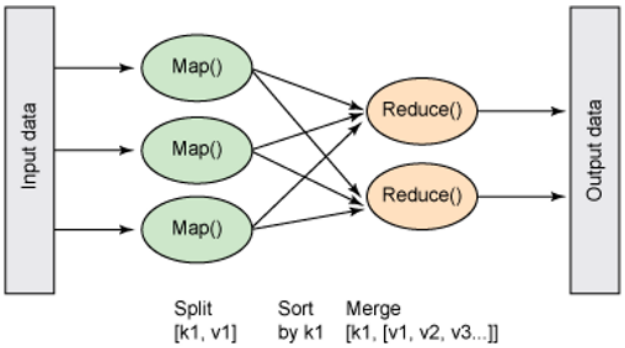

Figura 4: Etapas de MapReduce

MapReduce[10] es un paradigma de computación en paralelo que apareció en 2004 creado por Google[9] necesario para procesar grandes volúmenes de datos ante la necesidad en un futuro de tener cada vez mas cantidad de documentos a procesar desde varios equipos, de forma paralela para poder reducir sus tiempos de computo. Como podemos observar en la Figura 4, básicamente el programador solo necesita de dos etapas de computo para poder procesar los datos usando solo dos funciones `( Map y Reduce )`:

- **Map**: transforma un conjunto de datos de partida en pares (clave, valor) a otro conjunto de datos intermedios también en pares (clave, valor). Un formato, que hará más eficiente su procesamiento y sobre todo, más fácil su "reconstruccón" futura.
- **Reduce**: recibe los valores intermedios procesados en formato de pares (clave, valor) para agruparlos y producir el resultado final.

2.2. Apache Hadoop

Hadoop es un framework desarrollado por la Apache Software Foundation, como un conjunto de herramientas inspiradas en los documentos de Google para el procesamiento de BigData usando el paradigma de MapReduce y el Google File System.

Hadoop consiste básicamente en el Hadoop Common, que proporciona acceso a los sistemas de archivos soportados por Hadoop, que consiste en usar como sistema de ficheros: HDFS (Hadoop

Distributed File System). Un sistema de ficheros distribuido, escalable y portable, que almacena datos entre los distintos nodos, facilitando un gran ancho de banda en el conjunto del clúster montado. A diferencia de otros sistemas distribuidos, HDFS es altamente tolerante a fallos y está diseñado para ser usado en hardware de bajo coste.

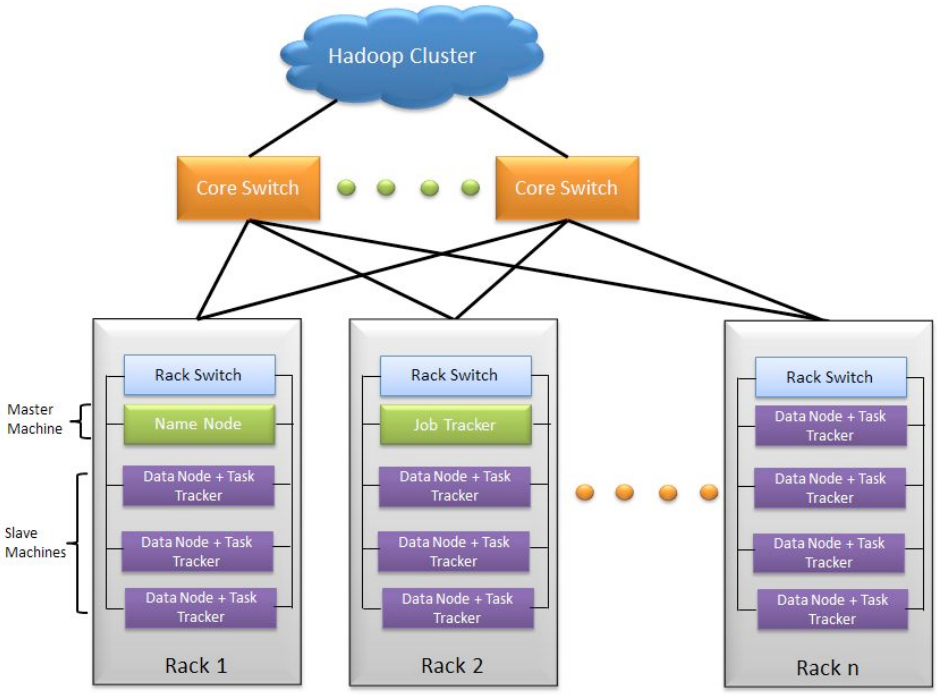

Figura 5: Distribución datos en Hadoop

En la Figura 5 podemos observar como la arquitectura de Hadoop distribuye los datos por los distintos nodos disponibles, permitiendo un equilibrio de carga distribuida de forma paralela, exceptuando un nodo centralizado llamado `Master Node`[11] el cual es el encargado de mantener la estructura de ficheros, que puede llegar hasta los *10 PB* de información almacenada. Está escrito en Java, también soporta los lenguajes de programación Python y Ruby.

2.3. Apache Spark

Desde 2014, aparece Spark, como un framework de computación en cluster open source basado en Hadoop en el que se obtienen mejores resultados en velocidad de calculo tanto en memoria, como en disco. Aprovecha el sistema de ficheros de Hadoop HDFS. En Spark se pueden utilizar varios lenguajes de programación, como son: Scala, Java, R y Python.
Podemos programar en archivos separados y lanzar las tareas hacía el clúster con `spark-submit`, o también nos permite interactuar con su shell interactiva `spark-shell` o `pyspark` en el caso de Python.
Además Spark ofrece muchos componentes extra dentro del conjunto de librerías dentro del propio framework como podemos observar en la Figura 6:

- Spark Core: Conjunto básico de librerías para poder trabajar con Spark
- Spark SQL: Permite la interacción con bases de datos SQL
- Spark Streaming: Permite la posibilidad de recolección de datos en streaming
- MLLib: Liberia para uso aplicado de Machine Learning
- GraphX: API para procesamiento de gráficos distribuidos

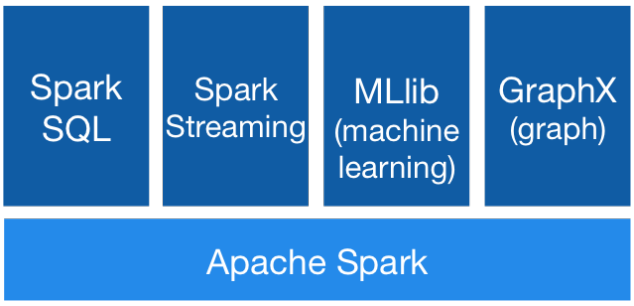

Figura 6: Componentes Apache Spark

2.3.1. Estructuras de datos en Apache Spark (RDD)

Conjuntos de Datos Distribuidos Resistentes (RDD)[12] es la estructura de datos que usa Spark. Es una colección de objetos distribuidos e inmutables. Cada dataset en un RDD, está dividido en particiones lógicas, las cuales pueden ser calculados en diferentes nodos del clúster. Los RDD pueden contener cualquier tipo de objetos en Python, Java o Scala, incluyendo las clases definidas por el usuario.

Un RDD es una colección particionada de registros de solo lectura. Los RDD se pueden crear a través de operaciones de transformación a partir de otros RDD o de datos que estén almacenados. Los Conjuntos de Datos Distribuidos Resistentes (RDD) son colecciones de elementos tolerantes a fallos que permiten ser tratados en paralelo ya sea con operaciones *iterativas* o *interactivas*:

Operaciones iterativas en Spark RDD

En la Figura 7 se muestran las operaciones iterativas de Spark RDD. Apache Spark, almacenará los resultados intermedios en una memoria distribuida (RAM) en lugar de almacenamiento estable (HDD) y hará que el sistema sea más rápido. Si la memoria distribuida (RAM) no es suficiente y se supera el limite disponible para almacenar los resultados intermedios, estos serán almacenados en el disco.

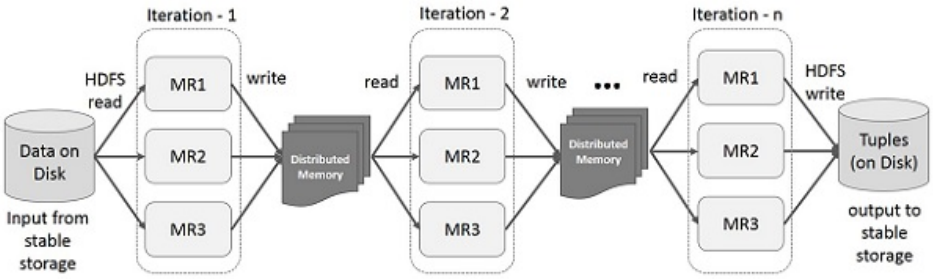

Figura 7: Operaciones Iterativas en Spark RDD

Operaciones interactivas en Spark RDD

En la Figura 8 se muestran las operaciones interactivas en Spark RDD. Si se ejecutan diferentes consultas en el mismo conjunto de datos repetidamente, estos datos pueden mantenerse en la memoria obteniendo mejores tiempos de ejecución.
De forma predeterminada, cada RDD transformado se puede volver a calcular cada vez que ejecuta una acción en él. Sin embargo, también puede persistir un RDD en la memoria, en cuyo caso, Spark mantendrá los elementos en el clúster para un acceso mucho más rápido la próxima vez que lo consulte. También se admite la persistencia de RDD en el disco o la replicación en varios nodos.

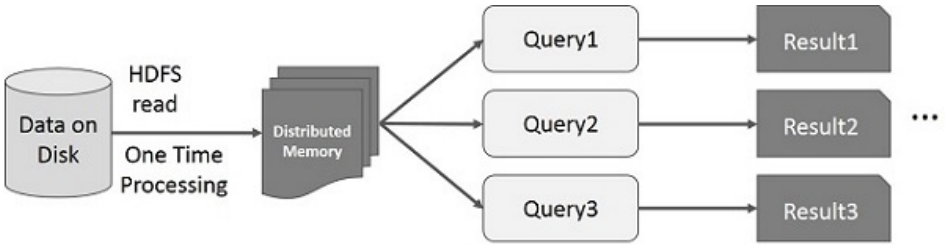

Figura 8: Operaciones Interactivas en Spark RDD

2.4. Apache Kafka

Apache Kafka[15] es una plataforma de streaming de datos que ofrece tres principales funcionalidades:

- Publicar y subscribirse a streams de registros, similar a una cola de mensajes, o un sistema de mensajeria empresarial.
- Almacenar streams de registros de forma duradera y tolerante a fallos.
- Procesar streams de registros mientras van ocurriendo.

Además Apache Kafka normalmente es usado en 2 tipos de aplicaciones:

- Construir pipelines de streams de datos en tiempo real para que se comuniquen de forma fiable entre sistemas o aplicaciones.
- Construir aplicaciones para la transformación de streams de datos en tiempo real.

Algunos conceptos a tener en cuenta que nos ofrece Apache Kafka que son beneficiosos sobre el resto de aplicaciones similares:

- Apache Kafka puede ejecutarse en un clúster tanto en uno como en más servidores, ampliando la capacidad hasta datacenters.
- Apache Kafka almacena los streams de registros en categorías llamados **topics**.
- Cada registro consiste en una *clave*, un *valor*, y un *timestamp*.

2.4.1. Envío y recepción de datos usando Apache Kafka

Primero tenemos que profundizar sobre como funciona la abstracción central en Apache Kafka que proporciona una secuencia de registros llamados *temas* (themes). Un *tema* es una categoría o nombre del feed al que se publican los registros. Los temas en Apache Kafka son siempre para múltiples suscriptores; es decir, un tema puede tener cero, uno o muchos consumidores que se suscriban a los datos escritos en él.
En la Figura 9 podemos observar como los productores publican datos a los temas de su elección. El productor es responsable de elegir qué registro asignar a qué partición dentro del tema. Esto se puede hacer de forma rutinaria simplemente para equilibrar la carga o se puede hacer de acuerdo con alguna función de partición semántica (por ejemplo, basada en alguna clave del registro).

2.4.2. Paralelización

Las instancias del consumidor pueden estar en procesos separados o en máquinas separadas. Si todas las instancias del consumidor tienen el mismo grupo de consumidores, entonces los registros se balancearán de manera efectiva sobre las instancias del consumidor tal como observamos en la Figura 10.

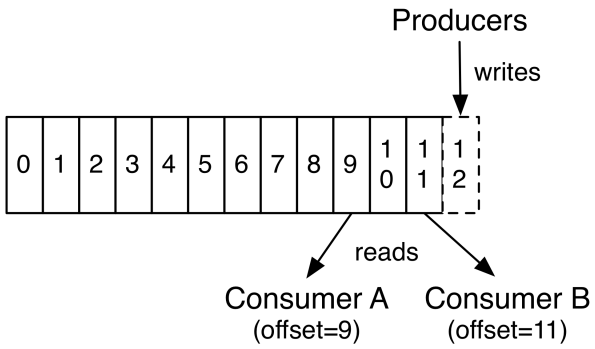

Figura 9: Envío y recepción de datos en Apache Kafka

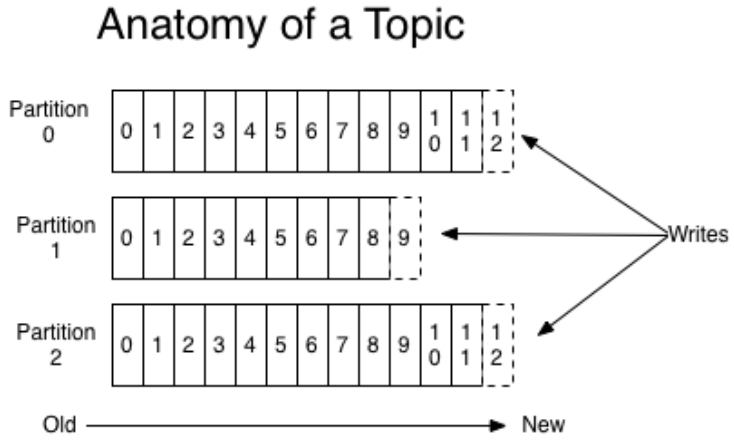

Figura 10: Replicación de topics en Apache Kafka

2.5. Apache Cassandra

La base de datos Apache Cassandra es un híbrido entre un modelo clave-valor y una base de datos tabular (Orientado a columnas), en la que es una buena elección cuando se necesita escalabilidad y alta disponibilidad si comprometer al rendimiento.

La escalabilidad lineal y la probada tolerancia a fallas en hardware básico o infraestructura en la nube lo convierten en la plataforma perfecta para datos de misión crítica. El respaldo de Cassandra para la replicación en múltiples centros de datos es el mejor en su clase, ya que proporciona una latencia más baja para sus usuarios y la tranquilidad de saber que puede sobrevivir a los cortes de energía regionales.

Características principales:

- Descentralizado: Todos los nodos del clúster tiene el mismo rol.
- Replicación: Sistema de replicación configurable para todo tipo de entornos.
- Escalabilidad: El rendimiento de E/S aumenta linealmente a medida que se añaden nodos.
- Tolerancia a fallos: Los datos se replican automáticamente a múltiples nodos.
- Consistencia: Se ofrece la elección del nivel de consistencia para E/S.

- Soporte MapReduce: Integrado con el sistema Apache Hadoop.
- Lenguaje de consulta similar a SQL: Introduce CQL (Cassandra Query Language)

2.6. Proyectos bioinformáticos existentes

Para poder elegir un proyecto base sobre el cual, basar nuestro estudio de mejora e implementación de una librería propia, lo primero que necesitaremos va a ser una búsqueda en internet de las aplicaciones bioinformáticas existentes en el mercado actual que utilicen las tecnologías requeridas para nuestros propósitos, en este caso como mínimo, que tengan alguna relación con Apache Spark, admitan formatos bioinformáticos y que su uso se pueda utilizar en entornos BigData.

Una vez hayamos encontrado un proyecto, se procederá a su análisis y prueba para poder ver si se adapta a nuestras necesidades, y en caso contrario, seguir buscando otros proyectos, de características similares, hasta encontrar un posible candidato.

A continuación se detallan varios proyectos que usen `Apache Spark` en el mundo de la bioinformática, los cuales se han intentado ejecutar, pero no se ha podido realizar ningún estudio sobre ellos, debido a que o han fallado en la compilación (usan versiones antiguas), o al ejecutar, no se han podido validar correctamente los datos y poder implementar alguna modificación para la creación de una librería. A continuación se muestran los proyectos analizados y los problemas encontrados.

2.6.1. MetaSpark

MetaSpark[6] es un proyecto creado en la universidad de Yunnan donde crean una utilidad basada en Spark para la lectura y referenciación de genomas. Intentan aprovechar los datasets proporcionados por Spark (RDD) de los cuales aprovechan la posibilidad de poder tener los datos en cache.

Sobre un fichero de 1 millón de lecturas, donde su tamaño es de 0,75 GB, se ve un aumento de rendimiento de un $\sim 4\,\%$. En otras pruebas se observa un aumento de escalabilidad y de rendimiento. El último commit que afecta a la librería es de la fecha del Octubre del 2016. Por tanto se puede esperar que no exista continuidad alguna al proyecto.

Se consigue:	**No se consigue:**
Compilar la versión 2 de MetaSpark.	Compilar la versión 1 de MetaSpark.
	Ejecutar la aplicación con los ejemplos facilitados por el autor.
	Ejecutar la aplicación con datos reales.

Tabla. 2: Tabla de éxitos en MetaSpark

Como no es posible realizar ninguna prueba con datos ya sean reales, o ejemplos proporcionados por el autor para poder analizar posteriormente los resultados obtenidos, descartamos este proyecto.

2.6.2. SparkSeq

SparkSeq[7] es una aplicación prototipo para análisis de ARN / ADN-Seq con precisión de nucleótidos en la nube. El objetivo del proyecto es crear una herramienta escalable y extremadamente rápida para estudios interactivos de ARN / ADN. Este proyecto está impulsado por Apache Spark, un motor rápido y general para el procesamiento de datos a gran escala, y Hadoop-BAM, una biblioteca para la manipulación de archivos en formatos bioinformáticos comunes utilizando Hadoop MapReduce.

Procedente de la universidad Warsaw University of Technology en Polonia. Su último commit es lanzado en 2014, al revisar el resto del proyecto se observa que dispone de dos versiones del proyecto, una para Hadoop con versiones 1 y 2, y otra que utiliza la versión 1 de Spark.

Se siguen las instrucciones que ofrecen en su repositorio para la compilación del proyecto en Spark, y no se llega a compilar con las versiones que disponemos de Spark, por lo que no podemos realizar ninguna prueba usando esta herramienta y por tanto, también queda descartado.

2.6.3. ADAM

ADAM[14] es una herramienta que puede ser usada como librería o por línea de comandos mediante el uso de Apache Spark para paralelizar el análisis de datos genómicos en entornos de clúster / computación en la nube. ADAM utiliza un conjunto de esquemas para describir secuencias genómicas, lecturas, variantes y genotipos. También se puede usar con datos en formatos de archivos genómicos heredados como SAM / BAM / CRAM, BED / GFF3 / GTF y VCF, así como también como datos almacenados en el formato de Apache Parquet columnar. En un solo nodo, ADAM ofrece un rendimiento competitivo para herramientas optimizadas de subprocesos múltiples, al tiempo que permite escalar a clústeres con más de mil núcleos. Las API de ADAM se pueden usar desde Scala, Java, Python, R y SQL.
Inicialmente podemos ver que es un proyecto reciente y con mucha actividad, ya sea en numero de colaboradores(63 colaboradores en GitHub), como en numero de commits (1774), el último en Abril del 2018.

Descargamos y compilamos la última versión, siguiendo las instrucciones facilitadas por el autor en el Listado 1, utilizando la última versión de Spark, pero este falla al compilar.

```
$ mvn install
```

Listado. 1: Compilación de ADAM

Tras buscar soluciones a la compilación, se encuentra que hay que omitir los test en el momento de compilar con Maven ya que dan fallos con las últimas versiones, por lo tanto hay que añadir la opción `-DskipTests` al compilar:

```
$ mvn install -DskipTests
```

Listado. 2: Compilación sin tests

Equipos utilizados para las pruebas en ADAM

Para realizar las pruebas sobre el funcionamiento y rendimiento que se obtiene con ADAM, se han utilizado dos equipos donde en la Tabla 3 se observan sus características. Uno es mi portátil, y el otro equipo es un clúster de la UDL (Pirgi).

	Portátil	**Clúster Pruebas UDL (Pirgi)**
CPU	Intel i5-430M (2 Cores / 4 Hilos)	Intel XEON CPU E5-2609 v4 (16 Cores)
RAM	4 GB	78 GB
HDD	120 GB SSD	2 TB with HDFS

Tabla. 3: Características equipos utilizados

Resultados obtenidos

Tomaremos como ejemplo un archivo de formato .bam cuyo tamaño original es de 3,7 Gb. que contiene un exoma del un humano que proviene del proyecto 1000 genome. Para poder ejecutar archivos de elevado tamaño, hay que ampliar la memoria ram disponible del driver tal como se observa en el Listado 3, ya que da error por superar el limite que tiene por defecto.

```
adam-submit \
--driver-memory 5g \
transformAlignments hdfs:///user/frb2/hdfs/HG0096.bam \
hdfs:///user/frb2/hdfs/HG0096_NOcache_OUT_UNCOMP \
-print_metrics
```

Listado. 3: Ejecución tarea simple con impresión de estadísticas

Después de ejecutar el comando del Listado 3 en ambos equipos, se puede observar que los tiempos entre un equipo y otro difieren bastante en la Figura 11, por lo que en el portátil no se realizarán mas pruebas debido a los altos tiempos obtenidos en comparación al clúster.

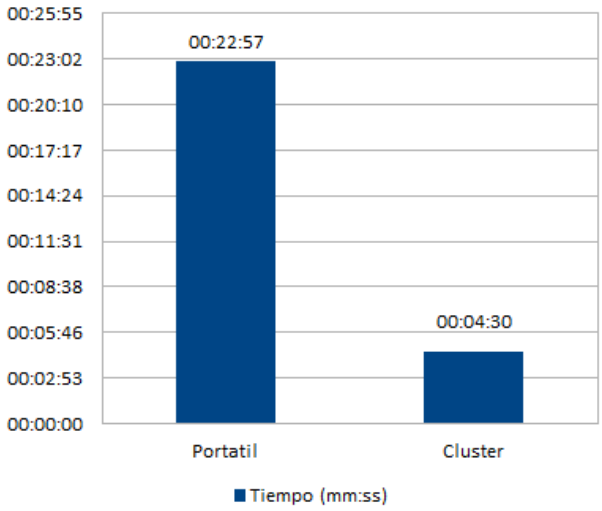

Figura 11: ADAM Conversion de alineamientos entre equipos

Se procede a realizar un análisis del coste según el numero de tareas a ejecutar, por lo que se obtiene un SpeedUp visible en la gráfica de la Figura 12.

Sintonización

En el clúster de pruebas, por defecto ya nos viene configurado con los valores optimizados para nuestro equipo. Pero para la instalación en otros equipos, hay que definir los recursos que se usarán en la ejecución de las tareas.
Para poder definir estos parámetros, se puede establecer mediante spark-submit al lanzar nuestra petición de tarea, donde le podemos asignar los valores que deseemos, o pasándole por primer parámetro en adam-submit para que este le pase los parámetros de Spark que le asignemos. Estos deben de definirse con doble guión (−−) tal como podemos observar en el Listado 4.

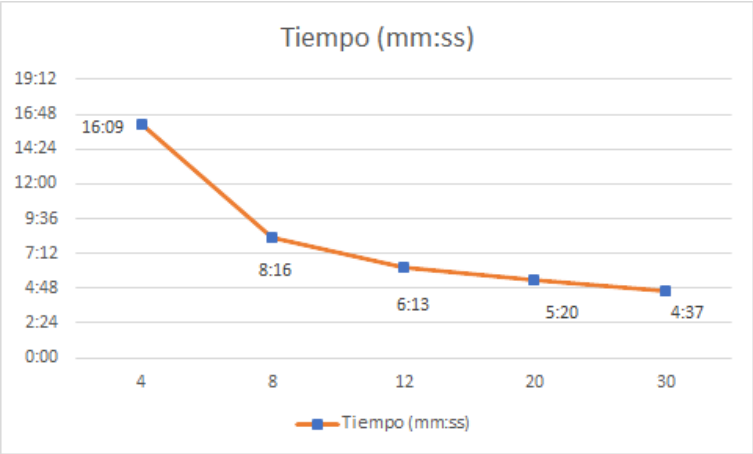

Figura 12: SpeedUp según número de tareas

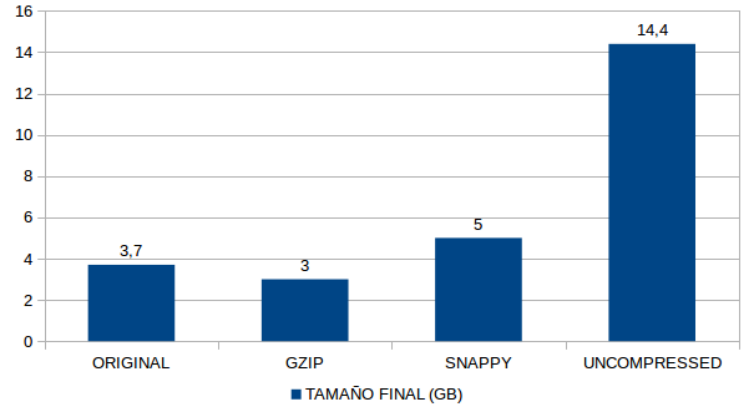

Figura 13: ADAM Alignments Size

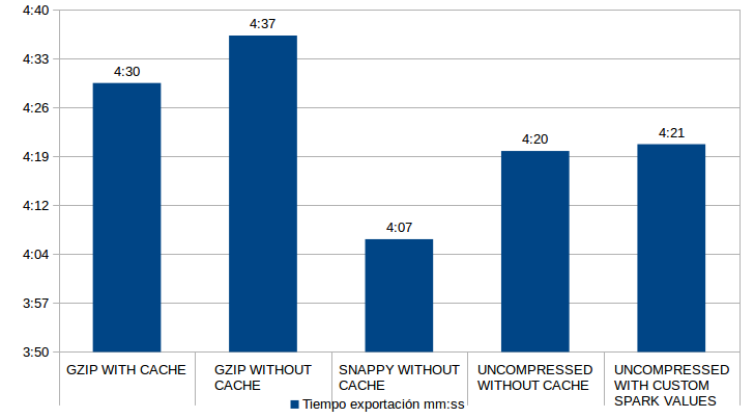

Figura 14: ADAM Tiempo de conversión de alineamientos en el clúster

```
adam-submit \
--driver-memory 5g \
--executor-memory 4g \
--executor-cores 4 \
--conf spark.ui.port=4050 -- \
transformAlignments hdfs:///user/frb2/hdfs/HG0096.bam \
hdfs:///user/frb2/hdfs/HG0096_NOcache_OUT_UNCOMP \
-print_metrics \
-parquet_compression_codec UNCOMPRESSED
```

Listado. 4: Ejecución personalizada

Procedemos a analizar los parámetros usados en la gestión de la tarea lanzada para Spark:

- –driver-memory 5g: Estableceremos un tamaño de 5 GB de memoria para el driver de comunicación, que será usado para poder tratar información entre los diferentes executors.
- –num-executors 15: Numero de workers que se ejecutarán. En nuestro caso estableceremos el valor de Cores totales del equipo - 1, que será usado como nodo principal.
- –executor-memory 4g: Memoria establecida para cada executor.
- –executor-cores 4: Núcleos del sistema que usará cada executor.

Conclusión sobre ADAM

Después de ejecutar varias pruebas y observar sus resultados tanto en la Figura 13 como en la Figura 14, donde podemos verificar si ADAM nos puede servir como proyecto base para poder realizar alguna mejora, se procede al estudio de su código. Se observa que el código de las funciones en general están muy fragmentadas en muchos ficheros, por lo que su seguimiento se hace difícil. Al llegar a las funciones básicas utilizadas para la lectura de ficheros en ADAM, se observa un uso de casts para llamar funciones en Scala y Java desde Python, por lo que solo las funciones intermedias están en Python, y las que gestionan la carga de ficheros y gestión de datos, se usan otros lenguajes. Al no poder extraer alguna función para poder analizarla y optimizarla, nos quedamos bloqueados y no podemos usar ADAM como proyecto base para su estudio, por tanto, también queda descartado.

2.6.4. BDT-Coffee

BDT-Coffee creado por Jordi Lladós (alumno de la UdL)[16] está basado en la integración sobre la consistencia de la información a través de bases de datos en Cassandra para T-Coffee[17], que previamente se han generado usando el paradigma de procesamiento MapReduce (PPCAS), haciendo que se permita el uso de grandes Datasets para ser procesados con el propósito de obtener mayor rendimiento y escalabilidad sobre el algoritmo original de T-Coffee creado por Cedric Notredame

Finalmente este será el proyecto por el cual se basará el TFG debido a que disponemos de soporte directo con el autor, y se ha implementado en los servidores que la UdL donde también tendremos acceso para su desarrollo, por el cual vamos a proceder a comentar sus aspectos importantes:

Arquitectura (T-Cofffe + Spark + Cassandra)

BDT-Coffee aprovecha varias tecnologías BigData que están conectadas entre si tal como se ve en la Figura 15, para poder aprovechar cada ventaja que ofrecen por separado y poder unirlas para ganar rendimiento. En el caso de BDT-Coffee, se utiliza T-Coffee como base para generar los datos, y en vez de gestionarlos en memoria como hacía originalmente, se aprovecha de Cassandra para poder generar la librería y guardarla en una base de datos, que nos permite liberar uso de memoria, y ofrece persistencia de datos.

Una vez se generan los datos en Cassandra, se llaman las tareas mediante Apache Spark, para permitir la ejecución de varios hilos de ejecución, para que vayan generando y obteniendo datos de Cassandra, y retornándolos hacia T-Coffee. Por lo tanto, en general BDT-Coffee está usando 3 tecnologías diferentes, que se interconectan entre ellas de forma eficiente y correcta.

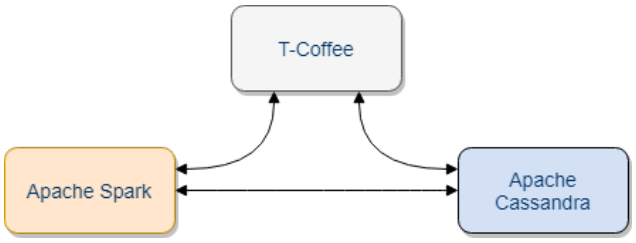

Figura 15: Arquitectura BDT-Coffee

Optimizaciones

Uso de Chunks: Cuando BDT-Coffee debe acceder a la librería primaria, y para evitar hacer muchas consultas a Cassandra, se utiliza un nuevo concepto chunk[22]. Este chunk es un entero que se pasa por parámetro a BDT-Coffee que permite agrupar datos en la base de datos optimizando el número de accesos a la misma. Además permite llevar un control de cómo se almacenan los datos en Cassandra y de cómo se recuperan tal como se puede observar en la Figura 16.

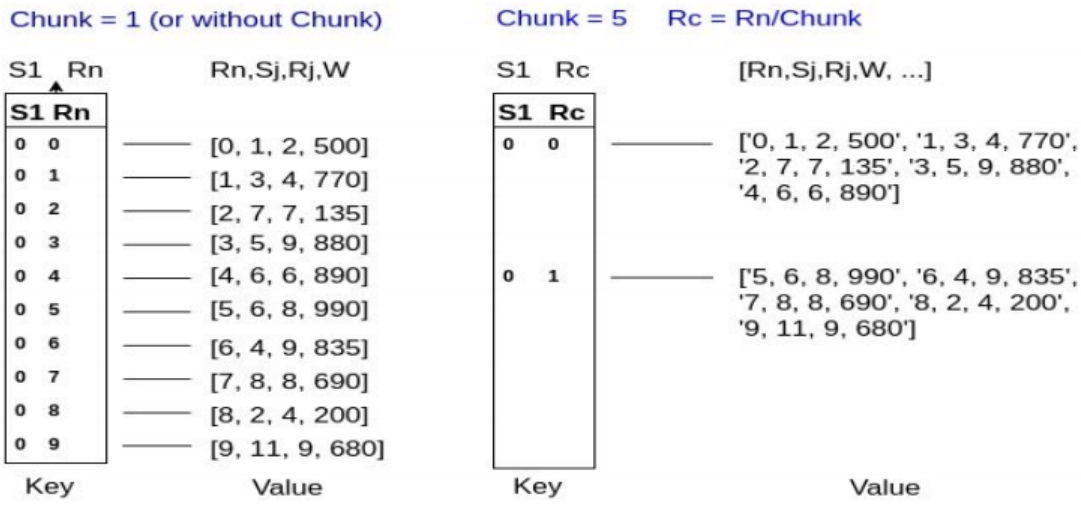

Figura 16: Organización de datos de Cassandra en Chunks

Cache de consistencia: BDT-Coffee utiliza una cache de consistencia para poder obtener mejores tiempos de respuesta, pero ésta, al ser un triple puntero de enteros similar a la de la Figura 17, consume mucha memoria en la máquina donde se ejecuta, por tanto lleva implementado un control de memoria llamado LRU que limita el uso máximo de memoria ejecutando una política de reemplazo de la cache. Por lo que se estudiará el eliminado de esta restricción, en la cual, quizás podamos obtener ampliar el uso máximo de memoria utilizada o ganar un incremento de velocidad en la gestión de memoria al prescindir de ella.

Prestaciones/Ventajas

`BDT-Coffee` aporta varias ventajas comentadas en el artículo de su autor[21], donde indica el problema que tiene `T-Coffe` sobre almacenar la librería de consistencia en memoria, y como con el uso de `Cassandra`, se solucionan los problemas de tamaño para almacenar la librería de consistencia, y de poder realizar alineamientos sin afectar al número total de secuencias a alinear.

La siguiente ventaja de `BDT-Coffee` sobre T-Coffee, es que se utiliza Apache Spark como motor para procesamiento de datos a larga escala en tiempo real, sobre una arquitectura Maestro/Esclavo. El cual, donde hay un coordinador central que se encargará de comunicarse con sus workers, donde

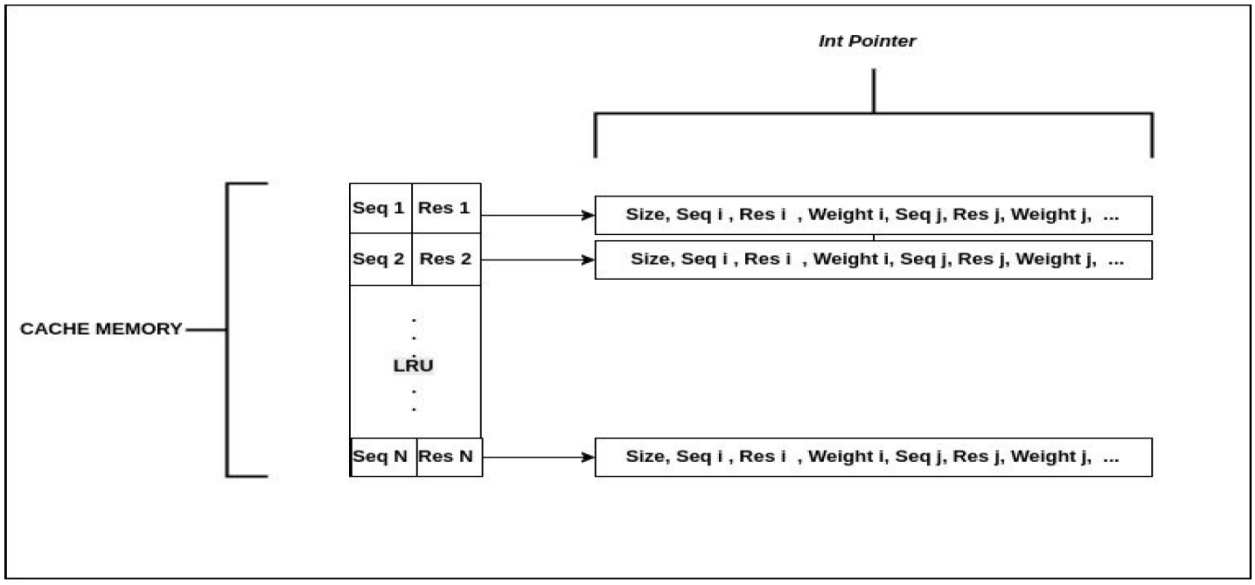

Figura 17: Estructura de la caché de consistencia

estos se encargarán de procesar los datos recibidos, permitiendo mejorar la eficiencia, al poder añadir mas nodos a la infraestructura y asegurarse que el tiempo de ejecución disminuye.

Desventajas (Librería extendida)

En la primera versión, no se usaba la librería extendida debido a que el tamaño generado era extremadamente grande (N^2L^2) para poder ser gestionada y guardarla en memoria. Provocando que el equipo que lo ejecutase se quedase sin memoria y diese un `Out of Memory`. Por tanto, el cálculo de las extendidas se debe realizar de forma bajo demanda con solo la información que se va a requerir en ese momento para el alineamiento, haciendo una carga dinámica sobre los datos a calcular y reduciendo el tamaño de la librería, pero sin tenerla completamente cargada.

2.7. Formatos de datos bioinformáticos

A partir de la NGS (Secuenciación de Siguiente Generación) han aparecido multitud de datos relacionados con las proteínas y los genomas. Por lo tanto cualquier dato que no se usa, este no estará debidamente analizado. ETL (Extracción, Transformación, Carga) es un paso importante en el análisis de las aplicaciones, en el que requiere un conocimiento de los pasos para tratar estos datos.[13]

- Entender las características de los datos
- Entender los datos
- Representar los datos
- Comprender las E/S de los datos al procesarlos.
- Resultados del análisis

2.7.1. Tipos de datos generales

CSV y TSV

CSV (*Comma separated value*) y TSV(*Tab separated value*) son formatos muy comunes utilizados para multitud de ámbitos, donde su contenido está en texto plano y sus campos están delimitados por el tipo de formato utilizado: comas en CSV, y tabulaciones para TSV.

Parquet

Parquet es un formato que almacena sus datos en columnas. Es bastante efectivo y proviene gran rendimiento en las consultas usadas en la plataforma Hadoop. Así mismo, Parquet está soportado en Apache Spark como soporte de almacenamiento para grandes volúmenes de datos bioinformáticos.

Formato de archivo de secuencia

Tal como aparece en la Figura 18, es un archivo de formato plano que consiste en parejas de key/-value de tipo binario. Son usados en MapReduce como archivos de entrada/salida. Temporalmente son la salida de la función mapeo en el paradigma MapReduce

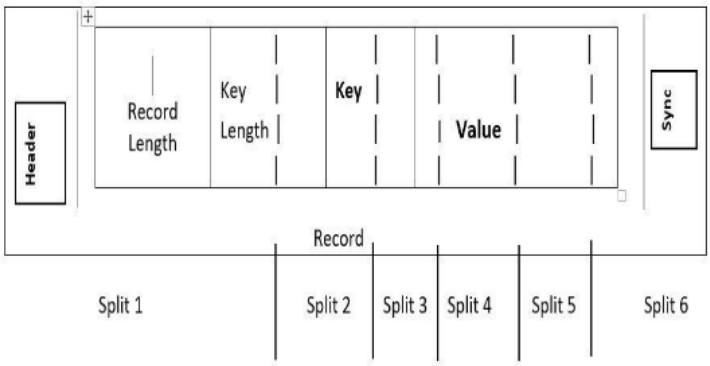

Figura 18: Sequance File Format

2.7.2. Tipos de datos bioinformaticos

FASTA Format

En el formato FASTA, su principal uso es almacenar información biológica de la secuencia de nucleótidos en las múltiples secuencias existentes. Este formato se caracteriza por empezar con el símbolo »"que determina el nombre de la secuencia y la salida acorde con su nombre. En la Figura 19, podemos observar un pequeño ejemplo de como se identifica la secuencia mediante una etiqueta, opcionalmente puede ir seguido de un pequeño comentario, y a continuación, se contiene todos los datos de la secuencia etiquetada.

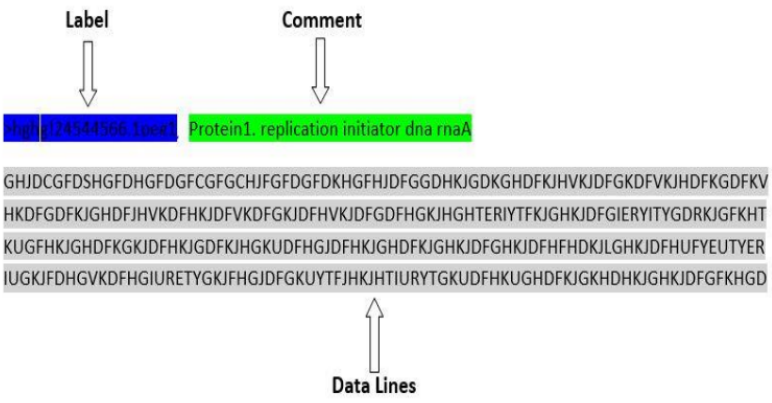

Figura 19: FASTA File Format

Fastq Fromat

Este tipo de formato es usado para almacenar ADN, RNA y datos de secuencias de genomas con su calidad. Como podemos observar en la Figura 20, la estructura de sus datos es similar a la de `FASTA`, pero en esta variante, se le añade una puntuación de calidad según los datos obtenidos en su análisis.

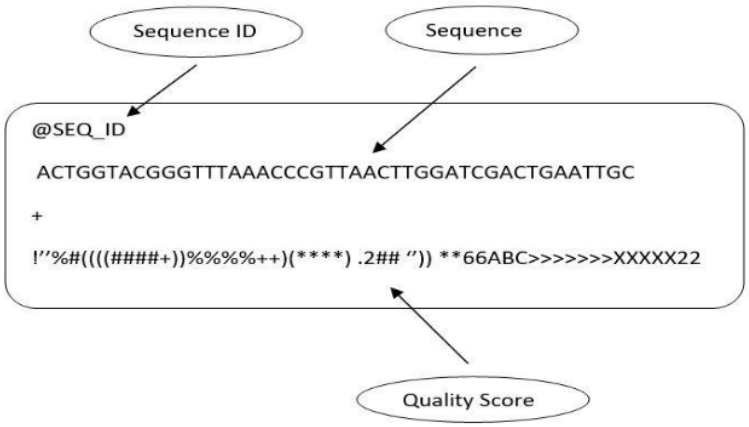

Figura 20: Fastq File Format

Formatos BAM (Binary Alignment Map) SAM (Sequence Alignment Map format)

BAM: Formato usado para el almacenamiento de datos en formato binaro sobre una secuencia ya sea alineada o no. Opcionalmente puede estar comprimido para ahorrar espacio. Su equivalente en formato de texto plano es el formato SAM. Los formatos BAM y SAM están diseñados para almacenar la misma información. Mientras que SAM esta pensado para su lectura fácil en texto plano, BAM al ser binario, permite una mejor compresión para la reducción de tamaño.

PHYLIP (multiple sequence alignment format)

El formato de archivo PHYLIP [20] almacena una alineación de secuencia múltiple en forma de árboles evolutivos (filogenias). El formato se definió originalmente y se utilizó en el paquete PHYLIP de Joe Felsenstein de la Universidad de Washington, y desde entonces ha sido respaldado por otras

herramientas bioinformáticas (por ejemplo, RAxML). En la Figura 21 podemos ver una secuencia tratada con PHYLIP.

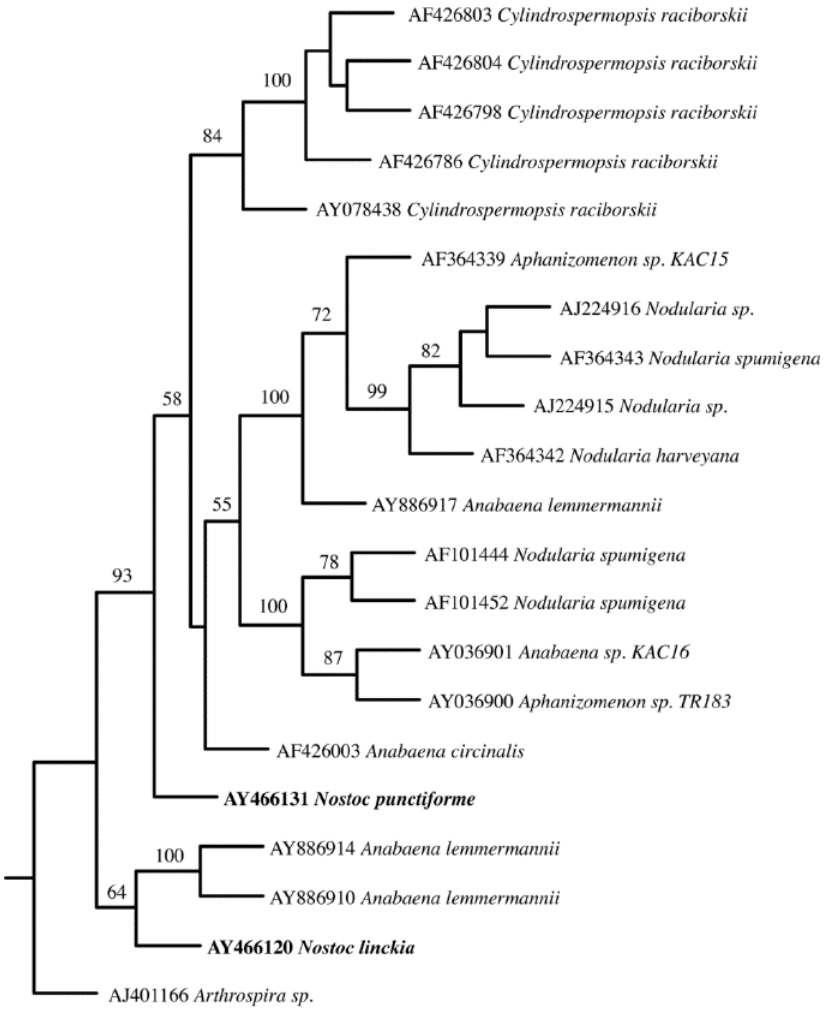

Figura 21: Árbol de una secuencia genómica en formato PHYLIP

3. Análisis y Diseño

3.1. Análisis

Esta tarea constituye la fase inicial en el desarrollo del proyecto la cual consiste en analizar las distintas tecnologías de las que disponemos actualmente para la implementación de cada uno de los componentes y decidir cuál es la que mejor se adapta y resuelve cada uno de los requerimientos que exige el proyecto en cuestión.

3.1.1. Lenguajes de programación en Spark

En la versión actual de Apache Spark 2.3.x a fecha de Agosto del 2018, los actuales lenguajes de programación soportados son: Scala, Java, Python y R, los cuales procedemos a analizar:

- **Scala**: Apache Spark está desarrollado en Scala de forma nativa. Scala es un moderno lenguaje de programación multi-paradigma diseñado para aprovechar los patrones de programación, la programación funcional y la programación orientada a objetos.
- **Java**: Java estaba soportado como lenguaje de programación nativo en Apache Spark, aunque actualmente dispone de su API en Java desde varias versiones.
- **Python**: Apache Spark dispone de una API para este lenguaje el cual tenemos un amplio conocimiento dado que durante la carrera se ha utilizado. Es capaz de aprovechar funciones y tipos de datos implementados en C o C++. Su principal debilidad es que se trata de un lenguaje interpretado, por lo que no usaríamos un lenguaje de programación de forma nativa.
- **R**: El lenguaje R[18] es un lenguaje de programación de código abierto mantenido por un equipo de desarrolladores voluntarios de todo el mundo. R es un lenguaje utilizado para realizar *operaciones estadísticas* y para la *generación de informes de análisis de datos*. Como su propósito no es el que deseamos y tenemos un escaso conocimiento del lenguaje **no lo utilizaremos** en nuestro proyecto.

Al disponer de 4 lenguajes, tenemos que elegir uno de ellos, por tanto hay que hacer una elección según las ventajas que nos aporten e ir descartando por los inconvenientes que nos puedan ocasionar. En caso de `R`, al ser un lenguaje utilizado para operaciones estadísticas, y generación de informes de análisis de datos, quedará descartado. `Scala` es el lenguaje nativo de Apache spark, pero al no haberlo tratado en la carrera, y personalmente tampoco se ha utilizado, también quedará descartado. Para la elección final entre `Java` y `Python`, elegiremos el segundo por su sencillez en el lenguaje, y por la facilidad de crear rápidos scripts sin necesidad de elaborar mucho código y necesitar mucho tiempo para su elaboración. Además al utilizar un *clúster BigData homogéneo* se puede implementar la librería con `Python + C` mediante `CTypes` para mejorar la eficiencia.

3.1.2. Lenguajes de programación para la librería

Para el desarrollo de la librería, podemos elegir cualquier lenguaje soportado en nuestros equipos de prueba, por lo que reduciremos el ámbito a lenguajes de baja complejidad como son `C/C++` y `Python`:

Python

Python es un lenguaje de programación fácil de aprender. Tiene estructuras de datos de alto nivel eficientes y un enfoque simple pero efectivo para la programación orientada a objetos, lo convierten en un lenguaje ideal para el desarrollo de scripts y de aplicaciones rápidas en muchas áreas de la mayoría de las plataformas ya que al ser un lenguaje interpretado proporciona la portabilidad necesaria en un entorno distribuido formado por máquinas heterogéneas.

C/C++

Tanto BDT-Coffee como T-Coffee usan C++ como lenguaje de programación base, además de ser un lenguaje que puede ser bastante flexible y donde el tiempo de ejecución es muy rápido por usar poco tamaño en memoria por cada proceso de ejecución. Como funcionalidad extra, tiene la facilidad para la creación y manipulación de estructuras de datos en bajo nivel, por tanto será un punto a favor en su elección. Existe una ventaja sobre *Python* y es que existe la posibilidad de exportar el binario como una librería compartida del sistema *(.so)* y poder ser ejecutada en el servidor *BigData* desde *Python* usando **ctypes**[19].

Por tanto la elección del lenguaje de programación para la librería que utilizaremos será `C/C++` compilándola como una librería del sistema `(.so)` y ejecutada desde `Python` usando `CTypes`.

3.2. Requisitos previos

Vamos a proceder a listar los requisitos previos al desarrollo de nuestra librería

- La librería ha de poder usar un lenguaje de programación compatible con Apache Spark.
- Ha de permitir la paralelización para poder reducir costes e incrementar rendimiento.
- Aprovechar el uso de infraestructuras horizontales como clústers.
- Proporcionar conectividad a la librería para ser usada con otros componentes.
- Permitir su instalación / distribución de la librería mediante una librería del so (.so)
- Permitir su uso de forma conjunta y compatible con BDT-Coffee.
- Se debe poder ejecutar en tiempo real bajo demanda cuando se necesite la librería extendida.

3.3. Diseño Librería

En esta sección describiremos el diseño de los componentes de los cuales se compone la librería y de como estarán interconectados entre sí. En la Figura 22 podemos observar un diseño general de los componentes que compondrán la librería en su estado final.

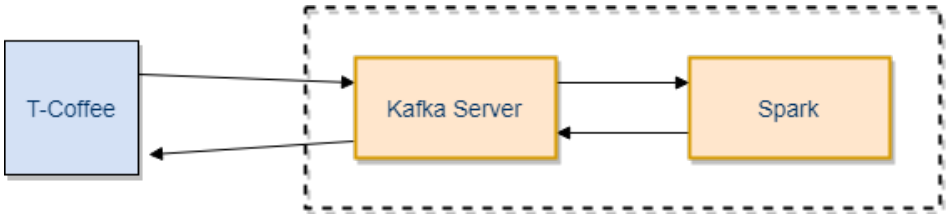

Figura 22: Componentes de comunicación

En los siguientes apartados del proyecto se describe como se realizará la adaptación del código original, para poder usar-lo en nuestra librería.

3.3.1. Extracción de funciones

Para el diseño de nuestra librería, tenemos que localizar en el código original el código responsable de calcular la librería extendida, así como analizar los requisitos de información (datos) necesarios para poder ejecutarlo (parámetros, librería básica, etc). En este caso nos fijamos en el archivo `evaluate.c` donde hay una función llamada **residue_pair_extended_score** que hace el calculo de *scores* (puntuación) sobre un par de secuencias extendidas, las cuales se pasan por parámetro. Por tanto podremos refactorizar esa función y extraer su código del proyecto base para utilizarlo en

nuestra librería. Al analizar la función, se observa que hace uso de una segunda función llamada **execute_query** la cual, también será necesario extraer su funcionalidad. En la Figura 23, podemos observar un esquema sobre la localización de toda la estructura base a extraer.

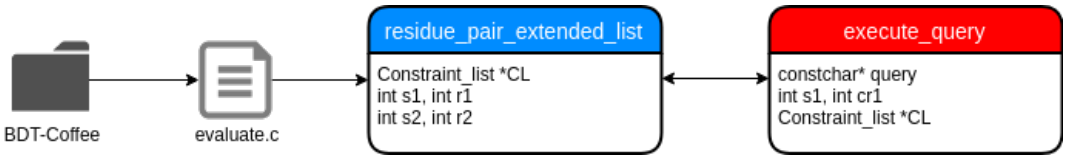

Figura 23: Extracción código BDT-Coffee

Originalmente en el fichero evaluate.c la función `residue_pair_extended_list` viene definida con 5 parámetros necesarios para introducirle y como retorno nos devolverá un `int` con el *scrore* obtenido del cálculo.

```
int residue_pair_extended_list ( Constraint_list *CL, int s1, int r1, int s2, int r2 )
```

En BDT-Coffee la función `residue_pair_extended_list` viene integrada con el resto de código de T-Coffe por lo tanto, usa las `Constraint_List` sin problema, por lo que este será uno de los primeros parámetros que eliminaremos de la función:

```
int residue_pair_extended_list ( int s1, int r1, int s2, int r2 )
```

El resto de parámetros que le pasaremos a la función serán los mismos que en BDT-Coffee, por tanto, al llamar la función y indicarle que secuencia se está tratando, no habrá problema de compatibilidad, y en el retorno, se le devolverá un *int* igual que ocurre con la función original, para mantener la compatibilidad completa entre proyectos.

3.3.2. Almacenamiento datos

BDT-Coffee aprovecha la estructura de datos heredada de T-Coffee donde incorpora multitud de datos referentes a las secuencias que tiene cargadas llamada:

Constraint_list *CL

Al buscar por el código original, buscaremos como está definida esta estructura de datos y extraeremos los datos mínimos necesarios para funcionar de forma independiente tal como podemos observar en el la estructura *Constraint*, donde almacenaremos en la estructura la secuencia S2, R2 y el peso obtenido. Esta estructura será usada posteriormente en una matriz donde almacenaremos la secuencia 1 (S1) como clave, y la estructura *Constraint* como un array de posibles secuencias conexas.

```
struct   Constraint{
         int s2;
         int r2;
         int w;
};
```

$$std :: vector < std :: vector < std :: pair < int, Constraint* >>> cache;$$

Para cargar los datos desde la base de datos de Cassandra, buscaremos las funciones relacionadas que se usan en **execute_query**, listadas en el Listado 5 y las copiaremos en nuestra librería respetando los nombres de las variables referenciadas entre ellas.

```
1  void print_error(CassFuture* future) {
2  ...
3  }
4
5  CassCluster* create_cluster(const char* hosts) {
6  ...
7  }
8
9  CassError connect_session(const CassCluster* cluster,CassSession* session
       )
10 {
11 ,,,
12 }
```

Listado. 5: Funciones relacionadas con Cassandra

Para la adaptación de la función `execute_query`, vamos a añadirle un parámetro nuevo llamado `cache_type` de tipo entero tal como aparece en el Listado 6, el cual nos indicará que cache queremos consultar. Esto nos será útil para la adaptación de las consultas a Cassandra, ya que como veremos más adelante, solo realizaremos dos consultas a la base de datos, y de esta forma le estamos indicando cual de las 2 caches estamos consultando.

El principio del código, se mantendrá casi intacto, pero manteniendo los nombres de las funciones anteriormente heredadas del Listado 5 para poder acceder a ellas y la eliminación de las referencias a la `Constraint_List CL`. Más adelante, a partir de la linea 44, es donde modificaremos los accesos a cache, respetando si es la cache *0* o *1* establecidas en el parámetro `cache_type`.

```
1  int execute_query(const char* query, int s1, int cr1, int cache_type) {
2
3    CassCluster* cs_cluster;
4    CassSession* cs_session;
5
6    CassError rc = CASS_OK;
7    CassFuture* future = NULL;
8
9    cass_int32_t r1;
10   cass_int32_t s2;
11   cass_int32_t r2;
12   cass_int32_t w;
13
14   int iStart = cr1 * Chunk;
15   int iEnd = iStart + Chunk;
16
17   std::vector<std::vector<tsl::sparse_map<size_t, int>>> scores;
18   std::vector<int> cache_it;
19
20   cache_it.resize(max_len,0);
21
22   if(!cs_session) {
23     cs_session = cass_session_new();
24     cs_cluster = create_cluster("127.0.0.1");
25
26     if (connect_session(cs_cluster, cs_session) != CASS_OK) {
27       cass_cluster_free(cs_cluster);
28       cass_session_free(cs_session);
29       return -1;
30     }
```

```
31    }
32
33    CassStatement* statement = cass_statement_new(query, 0);
34    future = cass_session_execute(cs_session, statement);
35    cass_future_wait(future);
36
37    rc = cass_future_error_code(future);
38    if (rc == CASS_OK) {
39      const CassResult* result = cass_future_get_result(future);
40      CassIterator* iterator = cass_iterator_from_result(result);
41      int size = cass_result_row_count ( result );
42
43      if(size==0) {
44        for(int i=iStart;i<iEnd && i<max_len+1;i++) {cache[cache_type][i].
             first = -1;}
45        cass_result_free(result);
46        cass_future_free(future);
47        cass_statement_free(statement);
48        return 1;
49      }
50
51      for(int i=iStart;i<iEnd && i<max_len+1;i++) {
52        cache[cache_type][i].second = (Constraint*) malloc(size * sizeof(
             Constraint));
53      }
54
55      while (cass_iterator_next(iterator)) {
56        const CassRow* row = cass_iterator_get_row(iterator);
57        cass_value_get_int32(cass_row_get_column_by_name(row, "r1"), &r1);
58        cass_value_get_int32(cass_row_get_column_by_name(row, "s2"), &s2);
59        cass_value_get_int32(cass_row_get_column_by_name(row, "r2"), &r2);
60        cass_value_get_int32(cass_row_get_column_by_name(row, "w"), &w);
61
62        cache[cache_type][r1].second[cache_it[r1]].s2 = s2;
63        cache[cache_type][r1].second[cache_it[r1]].r2 = r2;
64        cache[cache_type][r1].second[cache_it[r1]].w = w;
65        cache_it[r1]++;
66      }
67      cass_result_free(result);
68    } else {
69      print_error(future);
70      cass_future_free(future);
71      cass_statement_free(statement);
72      return -1;
73    }
74
75    for(int i=iStart;i<iEnd && i<max_len+1;i++) {
76      cache[cache_type][i].second = (Constraint*) realloc(cache[cache_type
           ][i].second, cache_it[i] * sizeof(Constraint));
77      cache[cache_type][i].first = cache_it[i];
78      cache_it[i] = 0;
79
80    }
81    cass_future_free(future);
82    cass_statement_free(statement);
83    return 1;
84  }
```

Listado. 6: Función execute_query para Cassandra adaptada

Chunk

Debemos tener en cuenta la variable llamada Chunk, esta será la encargada de definir el tamaño que una secuencia ha sido troceada en subconjuntos de N valores para cada S1. Esto es usado por cuestiones de espacio de tratamiento en memoria, ya que analizar toda la secuencia sin dividir, haría que los costes de computo aumentasen hasta saturar la maquina. Por tanto como podemos ver en la Figura 24, una secuencia, estará dividida en varias subsecuencias de N Chunks de longitud.

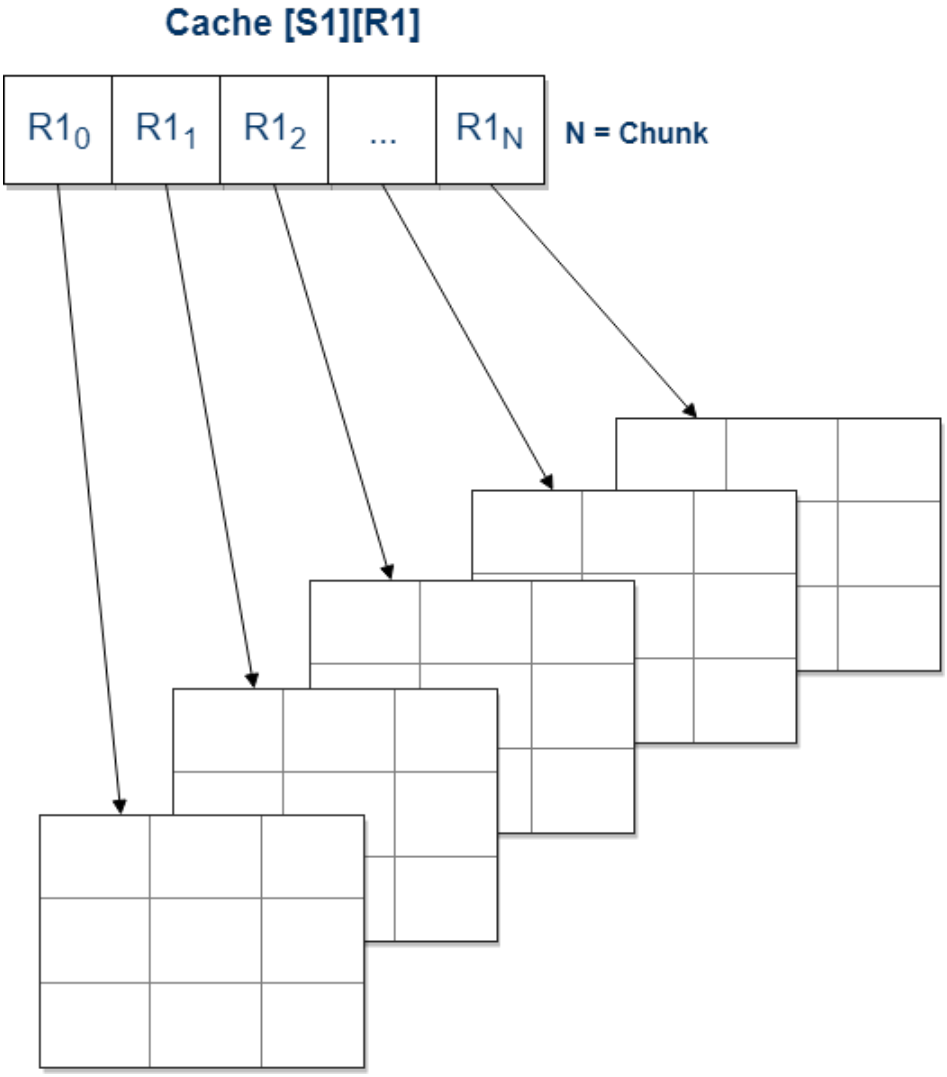

Figura 24: Diferentes caches por cada secuencia de N Chunks

3.3.3. Extracción del algoritmo de cálculo

Una vez ya tenemos acceso a los datos guardados en Cassandra, procederemos a la extracción del algoritmo de cálculo para poder aislarlo de BDT-Coffee. Analizamos la función `residue_pair_extended_list` del fichero `evaluate.c` tal como aparece en el Listado 7 y observamos que lo primero que necesitaremos será adaptar las llamadas a la función de Cassandra `execute_query` a las nuevas funciones sin pasarle las `Constraint_List`.

```
1 int residue_pair_extended_list ( Constraint_list *CL, int s1, int r1, int
      s2, int r2 ) {
2         if ( r1<=0 || r2<=0) {return 0;}
3
4         auto it = CL->scores[s1][s2].find(make_pair_un(r1,r2));
```

```
5           if(it != CL->scores[s1][s2].end()) {
6                   return it->second;
7           }
8
9           double score=0, max_score=0, max_val=0;
10          int t_s, t_r;
11          static int **hasch, max_len;
12
13          if ( !hasch || max_len!=(CL->S)->max_len) {
14             max_len=(CL->S)->max_len;
15             if ( hasch) free_int ( hasch, -1);
16             hasch=declare_int ( (CL->S)->nseq, (CL->S)->max_len+1);
17          }
18
19          if(CL->cache[s1][r1].first==0) {
20                  std::stringstream s;
21                  //s << "select s2,r2,w from ppcas." << CL->seq_name2 << "
                       _bdtc where key = ('" << s1+1 << " " << r1 << "','" <<
                        s2+1 << " " << r2 << "')";
22                  s << "select r1,s2,r2,w from ppcas." << CL->seq_name2 <<
                       " where key = '" << s1 << " " << r1/CL->qChunk << "'";
23                  execute_query(s.str().c_str(), s1, r1/CL->qChunk, CL);
24          }
25
26          if(CL->cache[s2][r2].first==0) {
27                  std::stringstream s;
28                  s << "select r1,s2,r2,w from ppcas." << CL->seq_name2 <<
                       " where key = '" << s2 << " " << r2/CL->qChunk << "'";
29                  execute_query(s.str().c_str(), s2, r2/CL->qChunk, CL);
30          }
31
32          for (int i=0; i<CL->cache[s1][r1].first;i++) {
33              t_s=CL->cache[s1][r1].second[i].s2;
34              t_r=CL->cache[s1][r1].second[i].r2;
35              hasch[t_s][t_r]=CL->cache[s1][r1].second[i].w;
36              max_score+=CL->cache[s1][r1].second[i].w;
37          }
38
39          hasch[s1][r1]=FORBIDEN;
40          for (int i=0; i<CL->cache[s2][r2].first;i++) {
41              t_s=CL->cache[s2][r2].second[i].s2;
42              t_r=CL->cache[s2][r2].second[i].r2;
43
44              if (hasch[t_s][t_r]) {
45                          if (hasch[t_s][t_r]==FORBIDEN) {
46                                  score+=CL->cache[s2][r2].second[i].w;
47                                  max_score+=CL->cache[s2][r2].second[i].w;
48                          } else {
49                                  double delta;
50                                  delta=MIN(hasch[t_s][t_r],CL->cache[s2][
                                       r2].second[i].w);
51
52                                  score+=delta;
53                                  max_score-=hasch[t_s][t_r];
54                                  max_score+=delta;
55                                  max_val=MAX(max_val,delta);
56                          }
57              } else { max_score+=CL->cache[s2][r2].second[i].w;}
58          }
```

```
59
60          max_score-=hasch[s2][r2];
61          clean_residue_pair_hasch ( s1, r1,s2, r2, hasch, CL);
62
63          if ( max_score==0) {return 0;}
64          else if ( CL->normalise) {
65              score=((score*CL->normalise)/max_score)*SCORE_K;
66              if (max_val> CL->normalise) {
67                          score*=max_val/(double)CL->normalise;
68              }
69          }
70
71          while(((sizeof(size_t) + sizeof(int) + 8) * score_elements + (
               sizeof(CL->scores) * (CL->S)->nseq)) > CL->max_mem*0.1) {
72               std::pair<int,int> tmp_key = lru_sc.remove();
73               score_elements-=CL->scores[tmp_key.first][tmp_key.second
                    ].size();
74               CL->scores[tmp_key.first][tmp_key.second].clear();
75          }
76
77          CL->scores[s1][s2][make_pair_un(r1,r2)]=(int)score;
78          lru_sc.insert(std::make_pair(s1,s2));
79          score_elements++;
80          return (int)score;
81          }
```

Listado. 7: Código original `residue_pair_extended_list`

La siguiente modificación que observamos, es la eliminación de la `LRU`, debido a que su uso únicamente es para la administración de memoria interna que utiliza `T-Coffee`, lo podremos quitar sin problemas. Otros métodos relacionados con `T-Coffee` que podemos eliminar, son la generación de pares `make_pair`, por tanto no habrá problemas para su extracción.

Como también crearemos nosotros nuestra cache, el codigo relacionado con la creación y modificación que hace `T-Coffee` sobre las caches, también la podremos eliminar, esto afecta a tanto a las extendidas, como a las que se usan desde la `Constraint_List`.

4. Desarrollo

En este capítulo, nos basaremos en realizar los diagramas necesarios para el desarrollo y la implementación en base a los resultados obtenidos sobre las tecnologías analizadas en el capítulo anterior.

4.1. Reducción de las Caches a generar

Para poder calcular el *score* extendido para un `pairwise ( S1 R1 S2 R2)`, necesitamos acceder a todas los residuos de *S1R1* y *S2R2*. Por tanto se necesitan 2 accesos a Casssandra y 2 entradas en la cache de consistencia para cada par, haciendo que el tamaño que requiere la cache original sea muy grande. Por lo tanto un punto de optimización que vamos a aplicar va a ser el poder reducir el computo de caches a 2 para cada par de secuencias. Donde tendremos una cache para la secuencia *S1R1*, y otra cache para la secuencia *S2R2*. Posteriormente en la Sección 4.3 vamos a diseñar las modificaciones necesarias del algoritmo original para el calculo de *scores* con las nuevas caches, de tal forma que primero se va a recorrer la secuencia 1 hacia la secuencia 2, y para el calculo de las secuencias extendidas, se van a hacer desde la secuencia 2, hacia la secuencia 1.

- 1º Calcular los pesos que hay desde la secuencia S1 a S2
- 2º Calcular los scores desde las secuencias extendidas de S2 hacia S1

Para ello, primero de todo, vamos a utilizar el método ya implementado en T-Coffee `cache.resize` para la modificación del tamaño de la cache.

$$cache.resize(nseq, std :: vector < std :: pair < int, Constraint* >> (max\_len + 1));$$

Donde las variables utilizadas se establecen como globales en la Figura 25, para generar una cache para solo *2* secuencias con un número máximo de *9999* elementos y un Chunk de *10* elementos:

```
int Chunk = 10;         // Tamaño del Chunk
int nseq = 2            // Numero de secuencias
int max_len = 9999      // Cantidad máxima de elementos
```

Figura 25: Definición número de secuencias

Para las consultas a Cassandra, vamos a incluir el parámetro `cache_type` de la función `execute_query` que hemos analizado en el capítulo anterior. Como son dos consultas donde se recolectaran los datos de Cassandra para copiarlos a las dos caches creadas, una para S1 *(0)* y otra para S2 *(1)*, se van a pasar los dos enteros para mantener un índice y saber cual de las dos caches estamos copiando.

```
1  ...
2  if(cache[0][r1].first==0) {
3    std::stringstream s;
4    s << "select r1,s2,r2,w from ppcas." << seq_name2 << " where key = '"
         << s1 << " " << r1/Chunk << "'";
5    execute_query(s.str().c_str(), s1, r1/Chunk, 0);
6  }
7  if(cache[1][r2].first==0) {
8    std::stringstream s;
9    s << "select r1,s2,r2,w from ppcas." << seq_name2 << " where key = '"
         << s2 << " " << r2/Chunk << "'";
10   execute_query(s.str().c_str(), s2, r2/Chunk, 1);
11 }
12 ...
```

Listado. 8: Modificación consultas hacia Cassandra

4.2. Generación array secuencias extendidas

Al estar extrayendo el código de T-Coffe para nuestra librería y que esta sea independiente, en la generación de la matriz de secuencias extendidas, es necesario crear nuestro propio código para poder adaptar el existente en T-Coffee. Primero se observa en T-Coffee como se declara la función y al ver que requiere de gran cantidad de funciones relacionadas con otros ficheros y que solo sirven de apoyo, se decide a crear la función desde cero y que sea lo más similar posible a la original de T-Coffee, de forma que es necesario adaptar el código original que aparece en el Listado 9 y su resultado es el que aparece en el Listado 10 donde se observa como queda el código después de adaptarlo.

```
1
2  #define DECLARE_ARRAY(type,wf,rf,function)\
3  type**  function (int first, int second)\
4    {\
5      return (type **)declare_arrayN (2,sizeof(type), first, second);\
6     }
7
8  DECLARE_ARRAY(int,write_size_int,read_size_int,declare_int)
9
10 int **declare_int2 (int f, int *s, int d)
11 {
12   int **r;
13   int a;
14   r=(int**)vcalloc ( f, sizeof (int*));
15   for (a=0; a<f; a++)
16     r[a]=(int*)vcalloc (s[a]+d, sizeof (int));
17   return r;
18 }
```

Listado. 9: Codigo original generación matriz secuencias extendidas

Para la adaptación del código, primero retiraremos la parte de las definiciones, ya que T-Coffee lo aprovecha para poder reutilizar código para varios tipos de datos. En la declaración eliminaremos la tercera variable *d* ya que no utilizaremos ningún desplazamiento. Para finalizar, retiraremos los *vcalloc* que son funciones creadas en *T-Coffee*, que en realidad son *mallocs*. Como nuestro objetivo es tener un array de `N*N valores` usando punteros, creamos el array como lo haríamos normalmente con *malloc* y recorriendo todos sus elementos para añadir la segunda dimensión.

```
1  int **declare_int (int f, int s)
2  {
3    int **r;
4    int a;
5    r=(int**)malloc ( f * sizeof (int*));
6    for (a=0; a<f; a++)
7      r[a]=(int*)malloc ( s * sizeof (int*));
8    return r;
9  }
```

Listado. 10: Generación matriz secuencias extendidas

4.3. Cálculo del score

Para el algoritmo que va a procesar los datos y generar un score, podemos observar su codigo final en el Listado 11 en el cual podemos ver que se hace un uso de las dos caches creadas previamente y el uso del array creado para las secuencias extendidas llamado `hasch`. También se fuerza el

uso de normalizar los datos, debido a que los datos obtenidos en primera instancia no corresponden con los que obtenemos con el código original de `T-Coffee`.

El resto del algoritmo será substituir en funcion de la secuencia analizada, que cache se está utilizando para obtener los datos, ya sea la `cache 0` o la `cache 1` equivalentes a la *Secuencia1* o la *Secuencia2*.

```
...
for (int i=0; i<cache[0][r1].first;i++) {
  t_s = cache[0][r1].second[i].s2;
  t_r = cache[0][r1].second[i].r2;
  hasch[t_s][t_r] = cache[0][r1].second[i].w;
  max_score += cache[0][r1].second[i].w;
}

hasch[s1][r1] = FORBIDEN;

for (int i=0; i<cache[1][r2].first;i++) {
  t_s = cache[1][r2].second[i].s2;
  t_r = cache[1][r2].second[i].r2;

  if (hasch[t_s][t_r]) {
    if (hasch[t_s][t_r] == FORBIDEN) {
      score += cache[1][r2].second[i].w;
      max_score += cache[1][r2].second[i].w;
    }
    else {
      double delta;
      delta = MIN(hasch[t_s][t_r], cache[1][r2].second[i].w);

      score += delta;
      max_score -= hasch[t_s][t_r];
      max_score += delta;
      max_val = MAX(max_val,delta);
    }
  }
  else {
      max_score += cache[1][r2].second[i].w;
  }
}
max_score -= hasch[s2][r2];

int normalise = 1;
int SCORE_K = 10;
if ( max_score == 0) {
  return 0;
}
else if ( normalise) {
  score=(((float)score * normalise)/(float)max_score)*SCORE_K;
  if (max_val > normalise) {
    score *= (double)max_val/(double)normalise;
  }
}
return (int)score;
```

Listado. 11: Algoritmo generación scores

Como adaptación y extracción de funciones, se hace necesario la definición de varias funciones incluidas en el Listado 12 que son declaradas en otros ficheros del proyecto y es necesario añadir,

para su correcto funcionamiento de forma individual referentes a las funciones MIN y MAX del cálculo del algoritmo.

```
#define FORBIDEN -1
#define MAX(x, y) (((x) > (y)) ? (x) : (y))
#define MIN(x, y) (((x) < (y)) ? (x) : (y))
```

Listado. 12: Definiciones extra

4.4. Compilación librerías auxiliares

Para poder compilar la librería y BDT-Coffee es necesario antes compilar las librerías necesarias para el correcto funcionamiento, ya que son requeridas para la compilación y su ejecución. En la Figura 26 se describe quien utiliza cada librería, y en los posteriores subcapítulos, vamos a describir como compilar cada librería por separado.

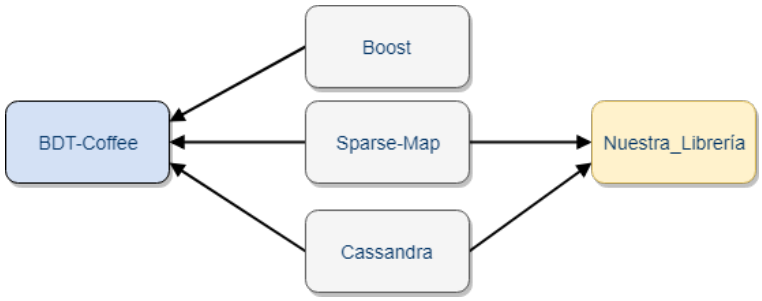

Figura 26: Librerías requeridas

4.4.1. Compilación Boost

El primer requisito para la compilación de BDT-Coffe es que este usa las librerías Boost para su compilación. Estas requieren ser descargadas y compilar su librería. La descargaremos en nuestra carpeta `$HOME` siguiendo las instrucciones del Listado 13.

```
$ cd $HOME
$ git clone https://github.com/boostorg/build.git
$ cd build
$ ./bootstrap.sh
$ ./b2 install --prefix=$HOME/boost/
```

Listado. 13: Instalación Boost

4.4.2. Compilación Sparse-Map

La segunda librería necesaria para la compilación de BDT-Coffee es `Sparse-Map`, esta librería, solo es necesaria su descompresión en nuestra carpeta `$HOME` tal como podemos observar en el Listado 14 ya que solo se necesitan sus cabeceras `(headers)`.

```
$ cd $HOME
$ git clone https://github.com/Tessil/sparse-map.git
```

Listado. 14: Instalación Sparse-Map

4.4.3. Compilación Cassandra

La tercera y última librería necesaria, es `Cassandra`, siguiendo el Listado 15, primero realizaremos la instalación para obtener sus headers necesarios para la compilación de BDT-Coffee, y posteriormente, generaremos una librería compartida (shared), necesaria para nuestra librería autónoma ya que el sistema destino no dispone de ella, este proceso también es valida si no disponemos de privilegios `root`.

```
$ cd $HOME
$ git clone https://github.com/Tessil/sparse-map.git
```

Listado. 15: Instalación Cassandra

4.5. Comunicación Python/C++

En *Python*, para poder comunicarse con la librería generada en c++ y obtener los resultados de sus funciones, se utilizará la librería `CTypes`[19] de *Python*, la cual, tenemos que editar el fichero de *C++* de nuestra librería para poder señalar que funciones van a admitir poder ser consultadas desde *Python* vía `CTypes`.
Substituir:

```
int residue_pair_extended_list ( int s1, int r1, int s2, int r2 )
```

Por:

```
extern "C" int residue_pair_extended_list ( int s1, int r1, int s2, int r2 )
```

4.5.1. Importación librería en Python

Una vez tenemos todos los requisitos previos para poder compilar nuestra librería, será necesaria la creación de un script en *Python*, para la importación del fichero *(.so)* generado, y poder llamar a la función definida como externa `residue_pair_extended_list` y obtener su respuesta como podemos observar en el Listado 16.

```
1  import os
2  import sys
3  import ctypes
4
5  sys.path.append('/home/frb2/outs')
6  # Antes de cargar nuestra libreria cargar cassandra
7  ctypes.cdll.LoadLibrary("outs/libcassandra.so.2")
8  libreria = ctypes.CDLL("outs/t_l.so")
9
10 # Variables de secuencias, kafka cargara estas variables
11 s1, r1, s2, r2 = 0, 1, 1, 1
12
13 # Preparacion variables
14 libreria.residue_pair_extended_list.argtypes = [ctypes.c_int, ctypes.
       c_int, ctypes.c_int, ctypes.c_int]
15 # Recepcion score
16 score = libreria.residue_pair_extended_list(s1, r1, s2, r2)
17
18 print "Score" + score
```

Listado. 16: Importación de la librería en Python

4.6. Comunicación T-Coffee/Spark

Para permitir la comunicación entre T-Coffee y Spark, usaremos Kafka como servidor de mensajería intermedio, siguiendo el esquema proporcionado en la Figura 27. De esta forma, Kafka generará los mensajes que podrán ser leídos de forma paralela por Apache Spark, con lo que se obtendrá un mayor rendimiento al poder paralelizar el calculo de resultados, debido a que es donde se necesita mayor tiempo de computo.

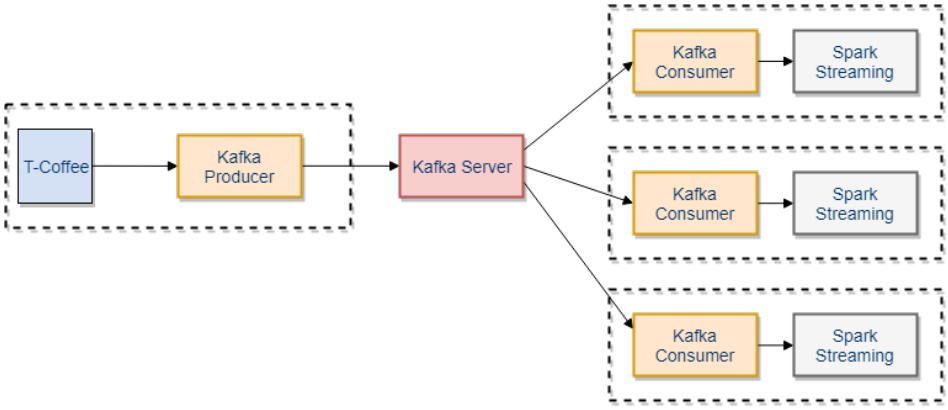

Figura 27: Envío de mensajes desde T-Coffee

4.6.1. Estableciendo protocolo de comunicación entre T-Coffee y Spark

Para poder identificar en que sentido van los mensajes, utilizaremos los *topics* para definir y restringir quien será el encargado de recibir y calcular los datos, y quien será el que identifique el mensaje como un resultado calculado. Para ello crearemos dos topics tal como podemos observar en la Figura 28, donde los comandos a ejecutar se observan al Listado 17.

Figura 28: Definición Topics Kafka

Donde:

- Secuencia: Será el mensaje enviado desde T-Coffee hacia Spark, el cual contiene las secuencias a calcular.
- Score: Contendrá en el mensaje un solo valor que será el resultado calculado en Spark el cual solo leerá T-Coffee.

```
$ kafka-topics.sh --create --zookeeper babel.udl.cat:2181 --replication-
    factor 1 --partitions 1 --topic secuencia
$ kafka-topics.sh --create --zookeeper babel.udl.cat:2181 --replication-
    factor 1 --partitions 1 --topic score
```

Listado. 17: Creación Topics en Kafka

4.7. Implementación

La implementación corresponde a la fase del proyecto en la que se lleva a cabo el desarrollo de los componentes a partir del diseño construido en la fase de análisis y diseño. Para hacer la implementación mas fácil, se ha construido un *Makefile* con distintas opciones tal como se observa en el Listado 18 para poder hacer mas fácil entender la construcción de la librería o la ejecución por separado de ella.

```
all:
        g++ l_cas.c -O3 -std=c++11 -I /home/frb2/cpp-driver/include/
        -I /home/frb2/sparse-map/ -lm -L /home/frb2/cpp-driver/build/
        -lcassandra -o l_cas -g
run:
        ./l_cas
lib:
        echo "export LD_LIBRARY_PATH=/home/frb2/cpp-driver/build/"
libreria:
        g++ l_cas.c -O3 -std=c++11 -I /home/frb2/cpp-driver/include/
        -I /home/frb2/sparse-map/ -lm -L /home/frb2/cpp-driver/build/
        -lcassandra -o l_cas.so -g -fPIC -shared
```

Listado. 18: Makefile de la librería

Donde cada opción del Makefile se utiliza para:

- **all:** Compilación de la librería como un **ejecutable** para poder ser ejecutada de forma independiente desde el terminal.
- **run:** Realiza la ejecución de la librería cuando está compilada como un ejecutable.
- **lib:** Muestra el comando para importar las librerías de Cassandra previamente compiladas.
- **libreria:** Compilación de nuestra librería como una librería del sistema operativo en fichero *(.so)* para poder ser importada desde *Python* mediante *CTypes*.

4.7.1. Asignación variables entorno

Como no se dispone de acceso root, hay que asignarle al sistema una variable de entorno para que pueda acceder a la librería que hemos compilado de Cassandra ejecutando la instrucción del Listado 19.

```
$ export LD_LIBRARY_PATH=/home/frb2/cpp-driver/build/
```

Listado. 19: Import Cassandra Libs

4.7.2. Creación datos librería Cassandra

Una vez ya tenemos los requisitos previos, ya podemos iniciar PPCAS para generar los datos que serán tratados posteriormente por T-Coffee. En nuestro caso se han generado manualmente los datos, ejecutando el script de creación de PPCAS tal como se indica en el Listado 20, de forma alternativa, se puede ejecutar el fichero `run.py` que envía la tarea mediante `spark-submit`.

```
PPCAS.py <Archivo_Secuencia> <N_Particiones> <IP Cassandra> <Chunk> <
   Nombre_salida>

$ python PPCAS.py ../examples/rrm_100 10 127.0.0.1 10 sdr_100
```

Listado. 20: PPCAS run

Una vez ejecutado el Listado 20 ya tenemos lista la base de datos Cassandra con los datos generados por la librería PPCAS, y donde serán accedidos desde la aplicación T-Coffee y nuestra librería.

4.7.3. Ejecución T-Coffee

Una vez completados todos los pasos previos, ya tenemos `T-Coffee` listo para poder ser ejecutado y realizar las pruebas necesarias para poder ver en funcionamiento `T-Coffee`. Siguiendo el Listado 21 ejecutaremos `T-Coffee` utilizando el mismo archivo que usamos en la Sección 4.7.2 para la creación de los datos en `PPCAS` y el mismo valor de *Chunk*.

```
$  ./t_coffee /home/frb2/BDT-Coffee/examples/rrm_100
        -extend_mode very_fast_triplet -dp_mode myers_miller_pair_wise
        -table_name rrm_100 -cassandra_seed 127.0.0.1 -chunk_size 10
```

Listado. 21: Ejecución manual de T-Coffee

Una vez ejecutado `T-Coffee`, nos aparecerá por pantalla que el calculo se está llevando a cabo, y podemos verificar que se ha podido compilar y ejecutar correctamente, para su posterior validación y finalización del proyecto.

5. Resultados/Experimentación

En esta sección nos vamos a dedicar a realizar diversas pruebas para poder validar y obtener diferentes resultados, así como comparar la versión original, con nuestra librería.

5.1. Validación

Para la validación, modificaremos el fichero `evaluate.c` para poder mantener las funciones antiguas y poder generar una comparación entre los resultados que nos genere la nueva función con la antigua.

```
1  int execute_query_original(const char* query,int s1,int cr1,
       Constraint_list *CL)
2  {
3  // Version del execute_query original
4  ...
5  }
6
7  int execute_query_nuevo(const char* query,int s1,int cr1,int cache_type)
8  {
9  // Version del execute_query nuevo
10 ...
11 }
12
13 extern "C" int residue_pair_extended_list_nuevo(int s1,int r1,int s2,int
       r2 )
14 {
15 // Version del residue_pair_extended_list nuevo
16 ...
17 }
18
19 int residue_pair_extended_list_original(Constraint_list *CL, int s1, int
       r1, int s2, int r2)
20 {
21 // Version del residue_pair_extended_list original
22 ...
23 }
24
25 int residue_pair_extended_list(Constraint_list *CL, int s1, int r1, int
       s2, int r2)
26 {
27   int score_original=residue_pair_extended_list_original(CL,s1,r1,s2,r2);
28   int score_nuevo = residue_pair_extended_list_nuevo(s1,r1,s2,r2);
29   if(score_original == score_nuevo)
30   {
31     printf("Score_O: %d - Score_N: %d \n", score_original, score_nuevo);
32   }
33   return score_original;
34 }
```

Listado. 22: Modificación codigo para la verificación

Después de aplicar los cambios del Listado 22, se observa que hay errores en la validación de la librería para gran número de pares de secuencias. Donde de cada *1000 pares* de secuencias, hay **177 pares** que su resultado es equivalente al código original, mientras que en **823 pares**, se obtiene un score incorrecto.
Tal como se observa en el gráfico de la Figura 29, hay solo un 18 % de resultados válidos, mientras que en las pruebas sobre un solo par, se consiguió obtener scores idénticos entre ambos proyectos,

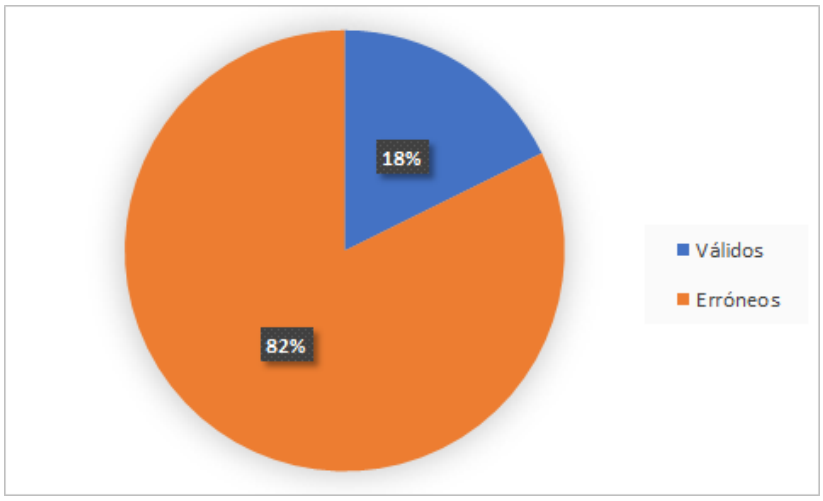

Figura 29: Porcentaje scores validados

por lo que se revisa el código en busca de posibles fallos de programación.

Tras la revisión del código, la única diferencia que se puede observar es que durante la versión de prueba para una sola secuencia, se utilizaban varios `printf` para debugear información en pantalla, y para la versión final se han eliminado. Tras restaurar y poner varios `printf`, se observa que si antes de retornar el score calculado, colocamos el siguiente `printf`:

```
//printf("Score: %lf Max_Score: %lf \n", (int)score, max_score);
```

el resultado retornado varía, haciendo que haya veces que incluso devuelva 0, por lo que es posible que exista alguna fuga de memoria en esos `printf`, que hagan que la conversión de datos sea incorrecta para su validación.

5.2. Resultados de Rendimiento

A continuación, realizaremos una ejecución entre los dos proyectos usando la misma muestra de datos rrm_100"para obtener diferentes resultados en tiempos de ejecución limitando el numero de pares a: `1, 10 y 100`, para poder ver como afecta el tiempo entre la llamada a `residue_pair_extended_list` y el tiempo que tarda hasta dar su score.

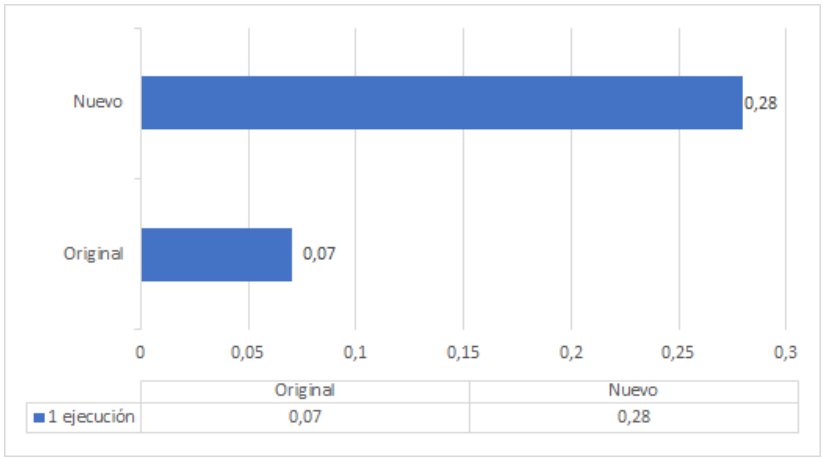

Figura 30: Tiempo ejecución para 1 muestra

Al ejecutar nuestro algoritmo para un solo par de secuencias, podemos ver que el tiempo de ejecución de la Figura 30, el tiempo que utiliza nuestro nuevo algoritmo, tarda 4 veces más que el código original, por lo que se procederá a incrementar el numero de pares a 10.

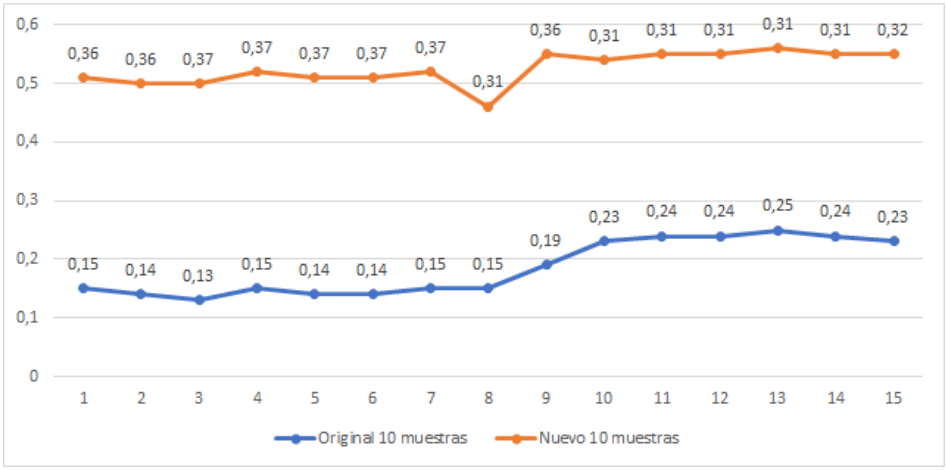

Figura 31: Tiempo ejecución para 10 muestras

Después de ejecutar 10 pares de secuencias y hacer su gráfico que aparece en la Figura 31, observamos que el tiempo de ejecución ha aumentado en los dos casos, tal como se indica en la tabla de la Figura 32, pero en el nuevo algoritmo sigue siendo más elevado que en el original, por lo que se decide hacer un último muestreo para *10.000* pares de secuencias y observar si en algún momento hay un cambio de tendencia.

Muestra	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Original 10 muestras	0,15	0,14	0,13	0,15	0,14	0,14	0,15	0,15	0,19	0,23	0,24	0,24	0,25	0,24	0,23
Nuevo 10 muestras	0,36	0,36	0,37	0,37	0,37	0,37	0,37	0,31	0,36	0,31	0,31	0,31	0,31	0,31	0,32

Figura 32: Tabla del tiempo de ejecución para 10 muestras

Al realizar la ejecución para *10.000* pares, se observa en la tabla de la Figura 33 que no hay cambio de tendencia en los tiempos de ejecución de nuestro nuevo algoritmo, por lo que se finaliza, realizando una recta de progresión en la Figura 34 para ver como va creciendo el tiempo de cálculo al aumentar el número de pares a calcular.

Muestra	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Original 10.000 muestras	0,32	0,35	0,33	0,33	0,35	0,35	0,37	0,36	0,88	0,96	0,91	0,99	0,92	0,81	0,94
Nuevo 10.000 muestras	1,21	1,24	1,25	1,24	1,26	1,25	1,3	1,46	1,4	1,49	1,64	1,53	1,55	1,47	1,8

Figura 33: Tabla del tiempo de ejecución para 10.000 muestras

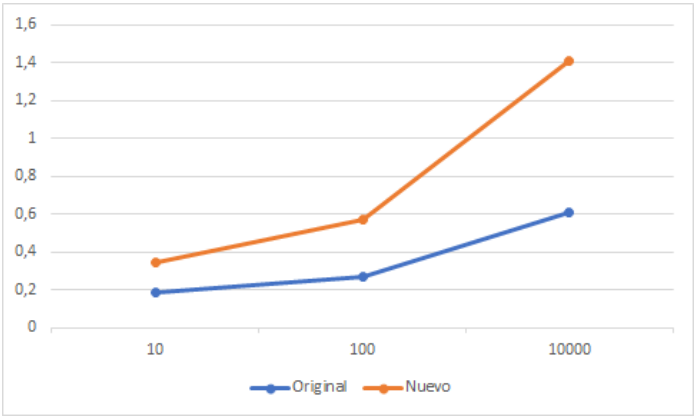

Figura 34: Recta de regresión según el número de pares calculados

6. Conclusiones y trabajo futuro

En esta sección final del proyecto se comentarán las conclusiones que se han obtenido al final del transcurso del proyecto, y comentar como se podría continuar desarrollando la aplicación para obtener mejores resultados.

6.1. Trabajos futuros

Dada la falta de tiempo para poder analizar y explotar las arquitecturas BigData disponibles, en un futuro, las primeras mejoras para el proyecto serían poder aprovechar la computación en paralelo que hemos diseñado con *Kafka* y el uso de *Streaming* de *Apache Spark* para poder implementarlo en código, debido a que en *Python* actualmente son tecnologías que están en constante cambio y no se pueden aprovechar con los ejemplos proporcionados en internet, ya que son de versiones anteriores no compatibles con las actuales y no se obtiene respuesta alguna en el envío y recepción de mensajes.

6.2. Objetivos alcanzados

Durante el transcurso de la elaboración de este proyecto se ha podido aprender sobre la elaboración de un proyecto desde cero, partiendo desde la búsqueda de las tecnologías disponibles, hasta poder obtener resultados con la aplicación diseñada, y poder documentarlo.

Como las tecnologías de BigData en el grado de informática no se experimentan, se ha empezado desde un nivel inicial de desarrollo. Por lo que en el futuro, puede que en el máster de informática *(ya que existe su rama de BigData)*, se pueda experimentar, y aprovechar todo el potencial que aporta `Apache Spark` y el resto de componentes para establecer aplicaciones *BigData* interconectadas entre varios componentes.

Dado que gran parte de la duración del proyecto se ha dedicado para el estudio del estado del arte, y no ha coincidido con la planificación inicial ilustrada en la Figura 3, donde hasta en la mitad del periodo de tiempo no se decidió optar por cambiar el trayecto del proyecto y usar `BDT-Coffee` como base para poder crear una librería, no se han podido alcanzar todas los objetivos iniciales establecidos para el código del proyecto final, pero estos se han tratado de forma teórica e individual para poder analizar y explorar todas las posibilidades que se podían usar en `Apache Spark`, `Kafka` y demás componentes relacionados que se han comentado en apartados anteriores.

Como observación, indicar que tal como se indicó en el principio del proyecto en la Sección 1.2.Motivación de la Página 3, donde se indicaba que todo el código estaría disponible para el público interesado en él, estará disponible tanto en GitHub en un repositorio público[23], como en el clúster `Babel` de la `UdL` el máximo tiempo posible.

$$https://github.com/yatan/TFG$$

Finalmente concluir que personalmente he disfrutado de todo lo aprendido durante el transcurso del proyecto y de haber descubierto nuevas tecnologías y arquitecturas sobre BigData y la computación en general.

Referencias

[1] IBM - ¿Qué es Big Data?
https://www.ibm.com/developerworks/ssa/local/im/que-es-big-data/

[2] Big Data: The 5 Vs Everyone Must Know Bernard Marr
https://www.linkedin.com/pulse/20140306073407-64875646-big-data-the-5-vs-everyone-must-kn

[3] Improvements on the previous technology EMBL-EBI
https://www.ebi.ac.uk/training/online/course/ebi-next-generation-sequencing-practical-cou
what-next-generation-dna-sequencing/improveme

[4] Learning Spark, Lightning-Fast Big Data Analysis,
By Matei Zaharia, Holden Karau, Andy Konwinski, Patrick Wendell,
O'Reilly Media, 2015.

[5] Advanced analytics with Spark, Ryza, Sandy,
ISBN 9781491912737,
O'Reilly Media, 2015

[6] MetaSpark: a spark-based distributed processing tool to recruit metagenomic reads to reference genomes.
https://github.com/zhouweiyg/metaspark

[7] SparkSeq: fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision.
https://bitbucket.org/mwiewiorka/sparkseq/

[8] Docker Imager with Jupyter Notebook Python
https://github.com/jupyter/docker-stacks/tree/master/all-spark-notebook

[9] MapReduce: Simplified Data Processing on Large Clusters
Jeff Dean, Sanjay Ghemawat Google, Inc.
https://research.google.com/archive/mapreduce-osdi04-slides/index.html

[10] Cuándo empieza esta era del Big Data: MapReduce
https://blogs.deusto.es/bigdata/cuando-empieza-esta-era-del-big-data-mapreduce/

[11] Massive Data Processing Part II (Temario del Master UdL)
Fernando Cores

[12] Apache Spark - RDD
https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm

[13] Modern Data Formats for Big Bioinformatics Data Analytics (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 8, No. 4, 2017
https://arxiv.org/pdf/1707.05364.pdf

[14] ADAM is a genomics analysis platform with specialized file formats built using Apache Avro, Apache Spark and Parquet.
https://github.com/bigdatagenomics/adam

[15] Apache Kafka® is a distributed streaming platform. What exactly does that mean?
https://kafka.apache.org/intro

[16] Scalable Consistency in T-Coffee through Apache Spark and Cassandra database - JCB2018
https://github.com/jllados/BDT-Coffee

[17] A collection of tools for Multiple Alignments of DNA, RNA, Protein Sequence http://tcoffee.org
https://github.com/cbcrg/tcoffee

[18] What is R Programming Language?
https://data-flair.training/blogs/introduction-to-r-programming/

[19] ctypes — A foreign function library for Python
https://docs.python.org/2/library/ctypes.html

[20] PHYLIP multiple sequence alignment format
http://scikit-bio.org/docs/0.5.0/generated/skbio.io.format.phylip.html

[21] Scalable Consistency in T-Coffee Through Apache Spark and Cassandra Database
Jordi Lladós, Fernando Cores, and Fernando Guirado
https://www.liebertpub.com/doi/10.1089/cmb.2018.0084

[22] TFM - Cálculo de la librería en Hadoop y su integración en T-Coffee
Oscar Ujaque Perez

[23] TFG - Diseño e implementación de una librería para soportar de forma eficiente tipos de datos bioinformáticos en Spark.
Fran Romero
https://github.com/yatan/TFG